DESIGN PARAMETERS FOR DISTRIBUTED PIM MEMORY SYSTEMS

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Masters of Science
in Computer Science and Engineering

by

Richard Cameron Murphy, B.S., B.A.

Peter M. Kogge, Director

Department of Computer Science and Engineering
Notre Dame, Indiana

April 2000

DESIGN PARAMETERS FOR DISTRIBUTED PIM MEMORY SYSTEMS

Abstract

by

Richard Cameron Murphy

Processing-In-Memory (PIM) circumvents the von Neumann bottleneck by combining logic and memory (typically DRAM) on a single die. This work examines the memory system parameters for constructing PIM based computers which are capable of solving significant problems. Using several data intensive benchmarks, simulations demonstrate that PIMs are capable of supporting enough data to be multicomputer nodes. Additionally, the results show that even data intensive code exhibits a large amount of internal spatial locality. A *mobile thread* execution model is presented that takes advantage of the tremendous amount of internal bandwidth available on a given PIM node and the locality exhibited by the application. Communication and other overhead is also discussed.

For my parents, Maura and Lester Murphy.

Sir Winston Churchill said, "the empires of the future are the empires of the mind." Thank you for founding my empire.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Peter Kogge, for his invaluable support and guidance in my pursuit of this work. At times, it seemed that the distance to the finish line halved with every revision (meaning it would never be reached)... thanks for pushing me over the line.

This work would not be possible without my committee, Dr. X. Sharon Hu, Dr. Jesús Izaguirre, and Dr. Edwin Sha. Their careful review is invaluable.

Dr. Jay Brockman and Dr. Vincent Freeh deserve many thanks for acting as the sounding board for some of my crazy ideas.

To my friends and colleagues for their continued support, encouragement, and (most importantly) distraction. In particular, Michael Niemier, Shannon Kuntz, Jeff Squyres, Ron Garcia, Jucain Butler, Kinis Meyer, and Jason Zawodny all helped tremendously.

Special kudos go to my good friend, Mr. F. Nicholas Rahaghi. Viva Las Vegas, Nick.

An early part of this effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under Cooperative Agreement number F30602-98-2-0180. The U.S. Government is authorized to reproduce and distribute for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements,

either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, or the U.S. Government.

CHAPTER 1

INTRODUCTION

Processing-in-Memory (PIM) systems (also known as Intelligent RAM [30], embedded RAM, or merged logic and memory) exploit tremendous amounts of memory bandwidth available for intra-chip communication, and therefore circumvent the von Neumann bottleneck, by placing logic and memory (typically DRAM) on the same die. This technology allows for the construction of highly distributed systems, but with a tremendous latency gap between high speed local memory macro accesses and remote accesses. Given an environment consisting of multiple PIMs, which in turn are made up of multiple nodes, memory macros of differing sizes and types, and various inter-connection schemes, the problem of managing the placement and movement of data though out the system is extremely complex. In an attempt to define and carefully balance the parameters of the construction of such a system, this thesis will enumerate and explore the design space of physically distributed PIM memory systems, and measure the effectiveness of potential implementations using realistic benchmarks. This, in turn, will reveal a new data management and execution model appropriate to PIM systems.

1.1 The Problem

The performance of modern computer systems is primarily determined by that of the memory hierarchy. The origin of this problem, and the cause of its continuous

increase, is the steady increase in the latency gap between processor and memory speeds. In fact, modern architectures go to Herculean efforts to circumvent the problem of accessing comparatively slow DRAM memories. For example, the next generation DEC Alpha (known as EV7), which is projected to run at 1 GHz, requires a startling 1.5 MB 6-way set associative level-2 cache **on chip** to provide the 16 GB/s bandwidth necessary for the processor to sustain performance [18]. Future Alpha and SPARC systems will both include 8 MB L2 caches on chip. These statistics do not take into account the memory overhead (in terms of tag and validity bits) for maintaining such a cache.

PIM seeks to eliminate this problem by providing the bandwidth via on-chip memory macros (typically DRAM) not to one central CPU but to many smaller ones distributed throughout the memory. (In fact the above Alpha very nearly qualifies as a PIM node by itself.) However, rather than providing a highly complex processor with caching, complicated branch prediction schemes, predication, and all the other relics of the high latency gap between the processor and memory, the PIM project seeks to construct simpler machines which take full advantage of the low latency access to local memory (through the use of simpler models such as vector units and multi-threaded execution), while providing high performance (potentially into the petaflop range) by working in tandem with other PIMs as part of a larger system. In this model, managing the placement, access, and movement of data throughout the system becomes the crucial element in delivering high performance. While the problem of accessing local data is easily solved, coping with remote accesses, which potentially incur many orders of magnitude longer latency than their local counterpart, becomes the deciding factor in the construction of effective multi-node PIM systems.

This may seem quite similar to the problem of a von Neumann machine ac-

cessing comparatively slow DRAM, however, there are three key differences: first, the envisaged PIM systems are parallel machines; second, since the processor and memory are merged, decision making can be made by any node in the system, not simply a hegemonic processor making dictatorial demands for data; and third, since multi-threaded execution has the potential for masking long latency remote memory accesses, PIM systems with such capabilities will be able to sustaining high throughput for programs which exhibit a high degree of parallelism.

Providing a clean, efficient, and easily programmable memory system on large PIM arrays is a unique challenge arising from the structure of PIM nodes. Unlike conventional nodes in a parallel machine, PIM systems are composed of relatively simple machines which possess small physical address spaces. A modern workstation, for example, will likely have 1 to 2 orders of magnitude more memory than a PIM node, as well as all the standard complex support systems necessitated by that amount of memory. Given this discrepancy, PIM systems potentially consisting of relatively large memory capacities will require many more nodes than conventional parallel systems of equal size whose nodes are typically much more powerful (closer to workstation size). Thus, there are generally two problems faced by the designers of such hierarchies: first, limited local resources reduce the amount of overhead information each PIM node can support; and second, the large number of nodes increases the complexity of interconnection networks and synchronization.

This thesis examines the construction of a global memory space for large PIM arrays with a variety of configurations using real applications to benchmark possible configurations and more fully enumerate the design space. The Data-Intensive Systems Benchmark Suite [4], with the addition of a molecular dynamics simulation, is used to examine access patterns on PIM arrays and determine the parameters for their construction.

1.1.1 Properties of Distributed Memory Systems

Generally speaking, distributed memory systems can be decomposed into at least two functions: first, a uniform method of naming resources (in this case, memory and processing resources); and second, a mechanism for communicating between elements of a heterogeneous memory hierarchy. The uniform naming convention requires that all names be treated equally. In a typical work station, "names" are virtual addresses and the hardware and operating system both contribute towards the translation of memory addresses. Moving data through various levels of the memory hierarchy also requires significant work by the hardware and operating system. Furthermore, as these memory systems typically present a flat memory hierarchy to the programmer, *transparency* and a uniform model of consistency must be maintained. Typical memory hierarchies (even on uniprocessor workstations) are extremely complex, yet that complexity is transparent to the programmer, in that the address space is still viewed as a large flat array. Furthermore, even though multiple nodes may be accessing the same piece of data, *consistency*, or the semantically correct value for that data as a function of time, must be maintained. Caches, for example, will provide transparency among hardware elements, while the operating system will typically require that pages which are to be accessed be present in physical memory (and not in secondary storage). From this simple discussion one can conclude that it is the function of a memory system to provide transparency and consistency. Typically such systems further provide protection from malicious or accidental access.

In a parallel environment, however, there is one further dimension which must be addressed, namely communication. Communication in a parallel environment can occur implicitly, as in the case of a memory request in NUMA, CC-NUMA, or COMA architecture, or explicitly as in the case of MPI, PVM, or Active Messages.

PIM systems should support both models of operations, however they are unlikely to support a *strict consistency model*, in which rigid semantics for the sharing and updating of data are supported, due to the complexity of communication required to update all copies of the data.

Given that there is a huge discrepancy between local and remote memory accesses (either implicitly or explicitly generated), this thesis will also examine the possibilities of masking the latency of such communication through the use of various techniques – most importantly architectural support for dynamic thread migration.

1.2 PIM Model

Assumptions regarding the configuration of the PIM arrays to be discussed are minimal. Since the purpose of this work is to determine the characteristics of the memory system of such an array, it is essential not to limit possible options by generating a large set of assumptions prior to experimentation. Thus, each PIM array will consist of multiple chips connected through an arbitrary interconnection network. Each chip will consist of multiple nodes (such as 4), which will, in turn, be capable of fast on-chip communication. Each node will have both processing resources and memory. The memory macros and processing resources for each node will be assumed to be fabricated using currently existing technology. A memory macro size of 2-32 MB per node will be assumed. The ND ASAP ISA [27] will be the assumed Instruction Set. Memory is assumed to be laid out in in the standard fashion (see Section 2.1.1, and Figure 2.1) – it consists of rows and columns. When an address is presented to the memory system, a row is read out, which is then presented to sense amplifiers and column decoders.

A PIM system could come in several forms (see figure 1.1): first, it could be an array consisting entirely of PIM nodes; second, it could be (a relatively small array)

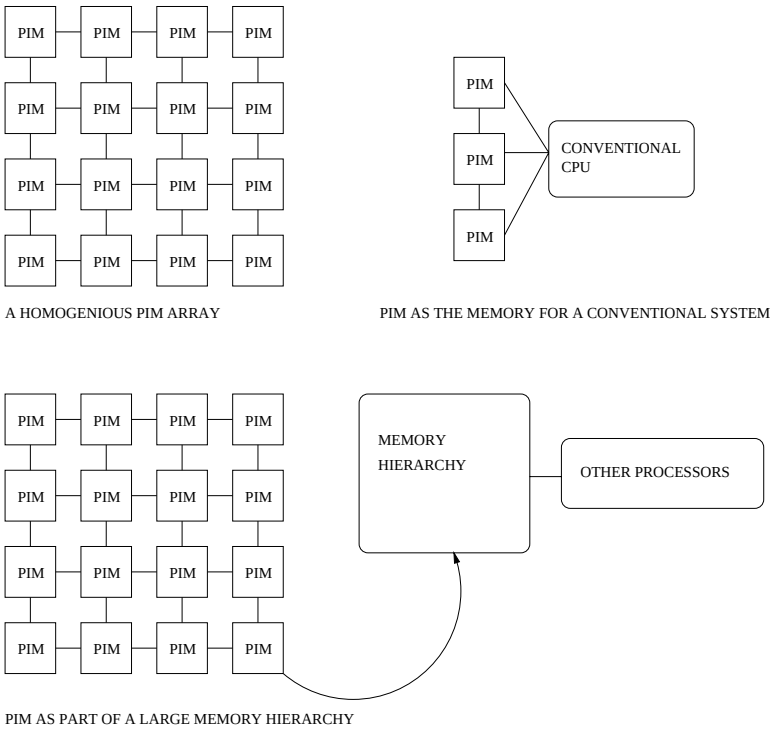


Figure 1.1. Types of PIM Systems

which constituted part of the memory hierarchy of a conventional processor; or third, it could constitute one or more levels of a hugely complex parallel machine's memory hierarchy. Given the diverging assumptions necessitated by any of the choices above, this work will concentrate on inter-PIM memory addressing and communication, rather than the interface between the array and another portion of a larger machine.

Generally speaking PIM arrays can be heterogeneous – they could consist of SRAM PIMs, DRAM PIMs, or PIMs of different generations with differing memory macro sizes and speeds. The analysis resulting from this work will be independent of these facts. Since the purpose of PIM is to utilize the on-chip bandwidth of a local memory macro (hence eliminating the von Neumann bottleneck), emphasis will be placed on the ability to effectively use local memory. Since a full row of memory (approximately 2 K-bits) must be read during each memory access, and this full row can be reused at low cost, additional experimentation will be conducted in the

effective re-use of these rows. Thus, the question of data placement will consist of several parts: placement within the array; placement within a given node; and placement within an "open row" and potentially even a "wide word" (256 bits).

The PIM communication mechanism is assumed to be a *parcel*. *Parcels*, as defined in [22] have the capability both of simple message-based communication, and of thread initiation (similar to active messages [36]). Since the system is multi-threaded, special attention will be made in regard to code access and loading, as well as the potential for covering memory access latency with threads. It should be noted that because parcels can contain the state of a running thread, they are **not** merely a light-weight RPC.

1.3 Thesis Organization

Chapter 2 will review the State of the Art in memory systems and inter-node communication for a variety of architectures. This will provide the necessary background for the analysis to come in later sections. Chapter 3 will review the benchmarks in detail. Their general properties, such as the types of memory accesses made (data, stack, or instruction fetch), row-buffer reuse and the treatment of the stack will be discussed, as well as the composition of a thread's context. Chapter 4 will review address space complexity in relation to the projected sizes of PIM nodes (particularly the physical address space). It will examine the trade-offs in page and PIM sizes as well as hardware complexity, paying particular attention to the memory costs.

Since there are very few constraints on the design of the memory system for this architecture (other than those previously alluded to – small simple nodes joined to large heterogeneous arrays which communicate through the use of *Parcels*), the first step in defining the design space is to understand thoroughly how data intensive

applications behave and what they require of the memory system. In addition to clearly enumerating the requirements of the memory system, the patterns of communication and the choices for data placement will be determined. Chapter 5 will examine local data set sizes and data placement in a first cut simulation using Shade. It will further lay the ground work for future experiments by determining the core code and data behavior to be analyzed.

Chapter 6 analyzes the communication patterns between a set of nodes during the "main loop" of the benchmarks. Using the information from Chapter 5, a data placement scheme will be used to traverse the essential data structures in the program against several interconnection schemes. The size of intermediate data sets will also be examined in detail, as well as the impact of moving threads from one node to another in search of data.

Chapter 7 will further build on the previous work by examining in detail the treatment of code, particularly in regard to threads and their context. Since the system will support the free movement of threads from one node to the next, extensive examination of what constitutes the state of a running thread as it moves will be made. Using the observations made previously about data placement, communication patterns, and the "context" of a thread, an execution model for *mobile threads* or *Rolling Snowballs* will be constructed. This execution model emphasizes fine grain threads and code mobility to help hide the latency of remote memory accesses or transfers from one level of the memory hierarchy to another.

To support the *mobile thread* environment, the idea of a *carpet bag cache* will be presented. This small, mobile cache will be used to effectively capture the context of a thread so that it can be moved (with low overhead) from one node to another.

Chapter 8 will summarize everything discovered in this work by recommending an organization for a single PIM node's memory system.

CHAPTER 2

A BRIEF REVIEW OF THE STATE OF THE ART

There is a large body of work devoted to the study of parallel computer architecture, and parallel memory systems in particular. Rather than focusing on innumerable implementation details, which are outside the scope of this work, this chapter will serve as a general guide to the memory schemes available in modern parallel machines. Additionally, it will provide background on both PIM and the Shade [34] system for program analysis.

2.1 Processing-In-Memory (PIM)

It has been said that it is more difficult for a man to get into heaven than for a camel to pass through the eye of a needle (Matthew 19:24). The von Neumann model of computer architecture has indeed made this a reality for data. Modern processors require tremendous amounts of data that the memory system is proving increasingly unable to deliver. The core of the problem is in the separate development of processing and memory technologies – processors, built around logic fabrication processes emphasize fast switching, while DRAM, which is typically constructed on a memory fabrication process emphasizes high density. The interconnection mechanism between the two is a relatively narrow bus, which cannot be greatly expanded due to the physical limit on the number of available pins and the high capacitance of inter-chip communication. This is known as the *von Neumann bottleneck*. Communication between entities on the same chip, however, suffers none of these problems

– it is fast, possesses large amounts of potential interconnection, and is relatively low cost. Recent developments in VLSI technology, such as the trench capacitor created at IBM, now allows for fabrication facilities which offer both high performance logic and high density DRAM on the same die.

Processing-in-Memory (PIM, also known as Intelligent RAM [30], embedded RAM, or merged logic and memory) exploits the development of this fabrication technology by creating devices which exploit the high bandwidth interconnection between processor and memory on the same chip. Several proposals for what the technology could achieve have been made. The Intelligent RAM (I-RAM) project at Berkeley seeks to place a general purpose core with vector capabilities along with a memory macro onto a die for embedded applications. Cellular phones, personal digital assistants, and other devices requiring processing power and small amounts of memory could benefit tremendously from this type of system, even if one only considers the potential advantages in power consumption.

Other, such as members of the Galileo group at the University of Wisconsin [5, 7, 6] see PIM as having tremendous potential for use in standard workstations. The on-chip memory macro becomes either a part of the memory hierarchy, or when the memory density becomes high enough, the entire memory hierarchy. Both of these views see PIM as a technology which fits very definitely into the framework of contemporary computer architecture, except that increased memory bandwidth is available. This can be seen in that each group proposes placing a fairly traditional (and expensive) processor core on the die. Naturally there are some additional abilities which can be exploited (such as the addition of vector instructions which can read the larger words available from memory.

The more radical view is that PIM is a technology which enables an entirely new computing model in which potentially millions of nodes can be utilized in tandem.

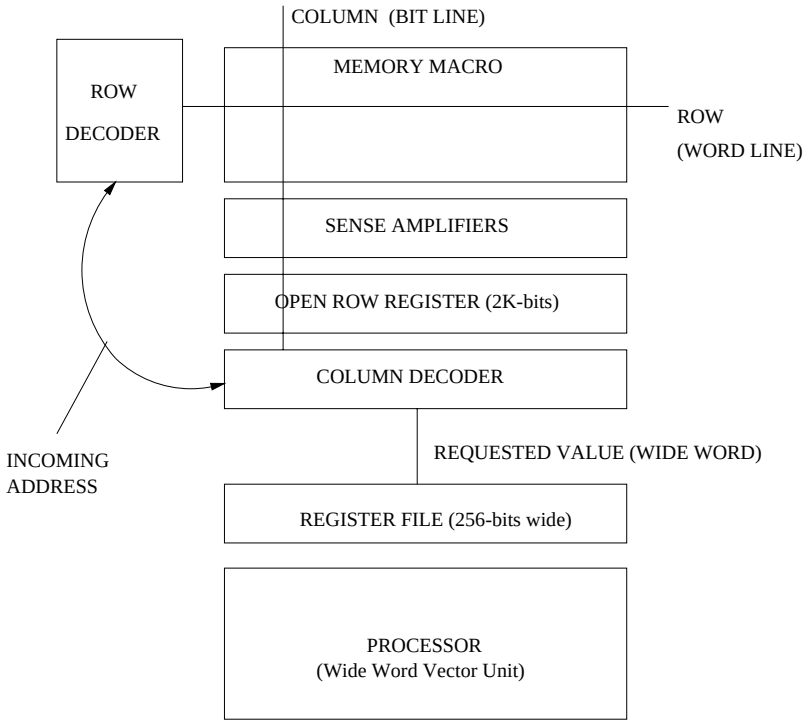


Figure 2.1. Typical PIM Memory Layout

2.1.1 Memory Layout: The Key to Big Bandwidth

As discussed previously, the key to overcoming the von Neumann bottleneck is by combining the processor and memory on the same die. Figure 2.1 shows a typical PIM layout. The Memory Macro, as in most memory systems, is laid out in rows and columns. When an address is requested (either for read or write), it is split into two parts: a row address and a column address. The row address is presented to a row decoder, and a signal to read out a particular row is generated on the appropriate *word line* (as indicated in the figure). The entire row (in this case 2K-bits) is presented to the *sense amplifiers* which increase the speed of switching by detecting small changes in the voltage on the *bit lines*. Each column has an associated bit line.

The value from the sense amplifiers is presented to the *open row register*, which

contains the *full row* from memory. At this point, part of the column portion of the incoming address is used by the *column decoder* to select which wide word (or "page") is read. Since a standard memory part would end here, only a very small portion of the full row can be presented to the memory bus (due to the limitation on the number of pins a chip can have). However, PIMs can take a much larger chunk of the full row (and even the enter full row during some operations). On chip logic, ie, the processor, is capable of requesting a *wide word*, in this case 256 bits.

Aside from the fact that a PIM is capable of requesting more data from memory than a normal processor, it is also capable of requesting that data at a significantly higher rate. By slightly enhancing the column decoder, multiple accesses to the same open row can be served at relatively low cost (since the row can be "latched in" on the first read).

2.2 Parcels

PIMs are assumed to communication through the use of *Parcels*. Though the semantics of a parcel are not fully defined, they are messages possessing intrinsic meaning which are directed at named objects. Rather than merely serving as a repository for data, they carry distinct high-level commands and some of the arguments necessary to fulfill those commands. Low level Parcels (which may be entirely handled by hardware) may contain simple memory requests such as: "access the value X and return it to node K." Higher level parcels are much more complicated. An example might be "begin execution of procedure Y with the following arguments and return the result to node L." Thus it should be assumed for the rest of this work that a Parcel is capable of performing both communication and computation, and it may be generated by the user, run-time system, or hardware for whatever mechanism may be appropriate.

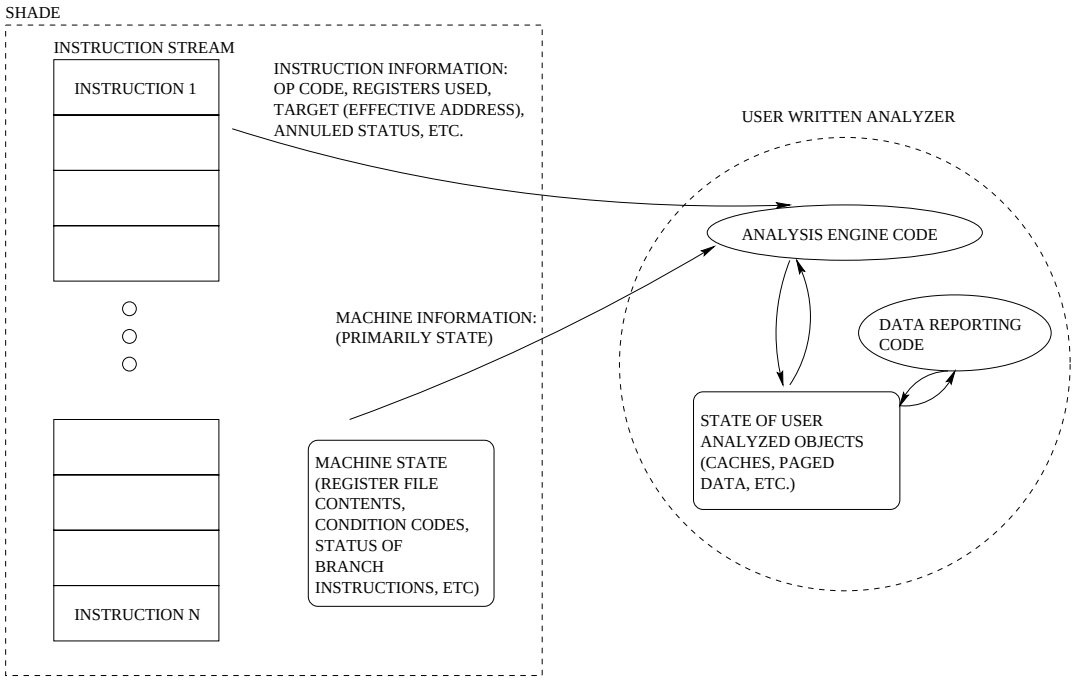


Figure 2.2. Shade Simulations

The fact that Parcels are directed at objects rather than physical nodes implies that a structure to determine *where* an object is located is essential to the successful construction of programs using Parcels.

2.3 Shade

The principle mechanism for benchmarking throughout this thesis will be the use of the Shade suite [34] developed by Sun Microsystems. The tool allows any SPARC binary to be analyzed architecturally in detail by providing a simple mechanism for analysts to write their own code to track the execution of an application. Shade provides information about every instruction which is executed, as well as the effects which that instruction had on the SPARC machine simulated.

As Figure 2.2 shows, programs are viewed simply as streams of instructions. Each simulator written for the purposes of this thesis uses those streams of instructions, combined with information Shade provides about the state of the machine, to

perform accounting for whatever is being tracked.

There are some key things which Shade **does not** do. It is incapable of tracing calls to the kernel, and therefore does not include accounting for system overhead. Generally, for the type of benchmarking performed in this thesis that is advantageous since only user code is of interest. Furthermore, Shade cannot be used to trace multi-threaded applications. The accounting needed to do so would require extensive kernel modifications. The code analyzed in this thesis is not multi-threaded, although some estimations as to the performance of threads are made later in the work. A package to allow Shade to perform accounting on simple run to completion threads is in the development process, however it was unavailable at the time of this writing.

2.4 DSMs and SMPs

The memory systems for parallel machines are often classified as either *Distributed Shared Memory (DSM)* or *Shared Memory Multiprocessors (SMP)*. Rather than exhaustively attempt to classify PIM systems, an informal definition for DSMs and SMPs will be offered. A DSM, such as the Rice TreadMarks system [3], generally consists of nodes which are tightly coupled with a piece of memory. The memory on each node is split into *local* and *global* parts, and the shared memory portion consists of the union of the global portion of each node's memory. Conversely, the nodes of an SMP, such as the Tera MTA [2], are tightly coupled. There are typically separate processing and memory nodes all connected via an interconnection network.

PIMs generally fit neither configuration in that while a node consists of both processing and memory elements (as in a DSM), it is conceivable that the processing capability of some nodes may consist of nothing more than servicing requests for memory (as in an SMP).

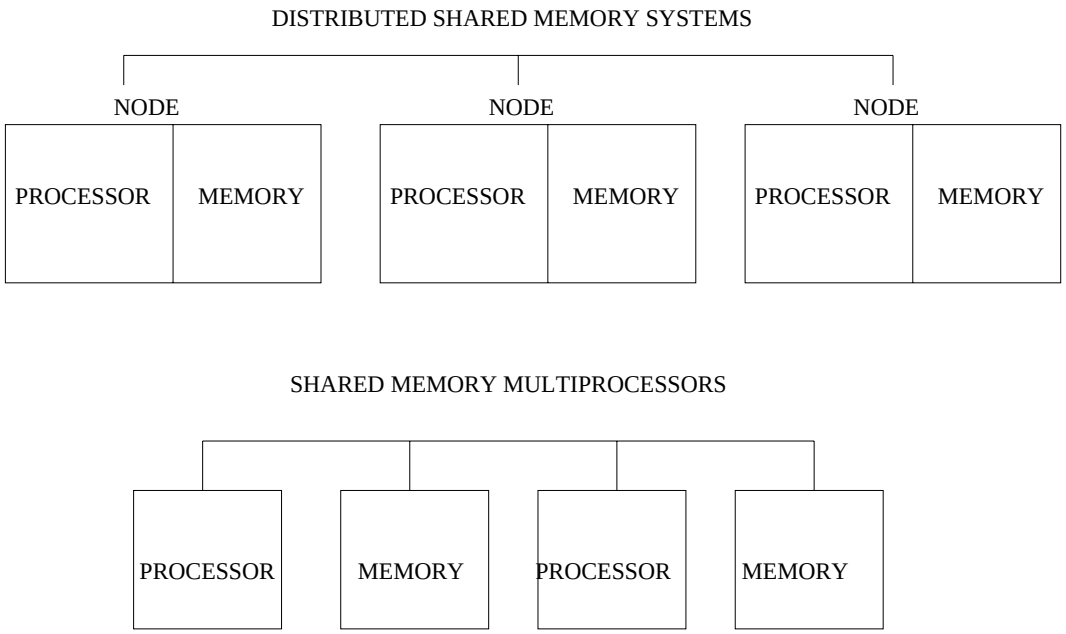


Figure 2.3. DSM and SMP Systems

2.5 NUMA and CC-NUMA Architectures

Non-Uniform Memory Access (NUMA) machines, as compared to Uniform Memory Access Machines (UMA), share perhaps the most in common with PIM systems, in the sense that local accesses occur quickly while remote accesses may encounter long latencies. The memory access on a PIM is non-uniform in the extreme. As previ-

Figure 2.4. A Typical CC-NUMA implementation

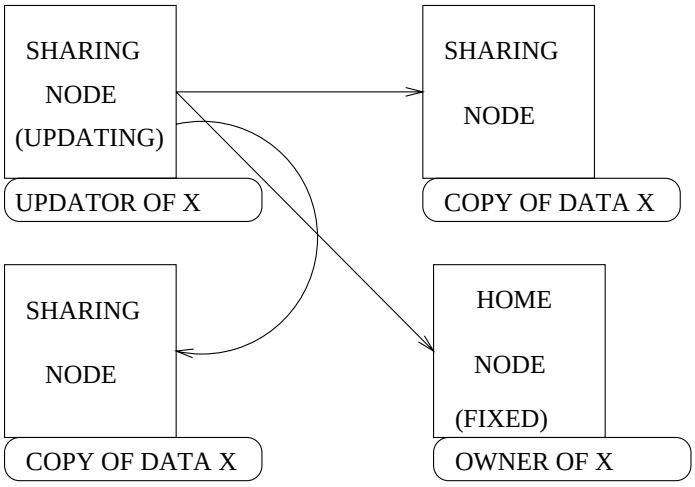
ously discussed, local accesses are extremely fast while non-local accesses are likely to be extremely slow. However, this is where the relationship grows increasingly distant. The Cache Coherent NUMA (CC-NUMA) architectures provide support for coherency amongst nodes in a NUMA system. That is, a *coherency protocol* is defined which ensures that each node contains the semantically correct value of the data which it is examining, which ensures that a consistent computation always takes place.

The Stanford FLASH [25] (which is a descendant of the DASH machine [26]) is a classic example of a CC-NUMA architecture. There is extensive research into both the coherency protocol used and the methods of efficiently storing information as to who is accessing a particular piece of data. The Scalable Coherency Interface (SCI) [21] is an example of a coherency protocol definition (as well as an interconnection network standard). The SCI uses linked lists to track who is accessing a particular cache line. Many other protocols use more expensive sparse matrices. It should be noted, however, that most CC-NUMA organizations do not scale beyond a few thousand nodes because such extensive information must be kept about data which is being shared.

Even though CC-NUMA machines track the sharing of data throughout the system, a given physical address always has a fixed location or *home node*. Although many nodes may be sharing a given address, or even updating that address, when the address is presented to the system it can deterministically be found by referring to the home node. See Figure 2.4 for details.

Figure 2.5 shows the updating of a piece of data under a CC-NUMA scheme.

The MIT Alewife [1], the Convex Exemplar and the SGI Origin are other examples of CC-NUMA machines.



NOTE: X represents a piece of data which must be updated. The HOME NODE owns X, but is not the node performing the update. The coherency updating protocol will determine the mechanism for in a semantically correct fashion.

Figure 2.5. A CC-NUMA machine during an update.

2.6 COMA Machines

In contrast to CC-NUMA architectures, the Cache Only Memory Architecture (COMA) allows for the actual migration of data throughout the system. That is, a given address does not have a fixed home node, rather the system must determine the home node by looking in a directory. Since the location of data is free to move about the system, addressing simply becomes a name space for data, rather than implying a fixed physical location. When an address is presented to the system, it is located through the use of a *directory*, and it may migrate accordingly. It should be noted that most COMA architectures support Cache Coherent protocols – that is, there are copies of various pieces of data throughout the system. The primary difference is that the "original" or "master copy" of data may migrate based on demand. See figure 2.6.

Figure 2.7 is an example of a COMA machine performing an update.

There are several examples of COMA architectures constructed in hardware or

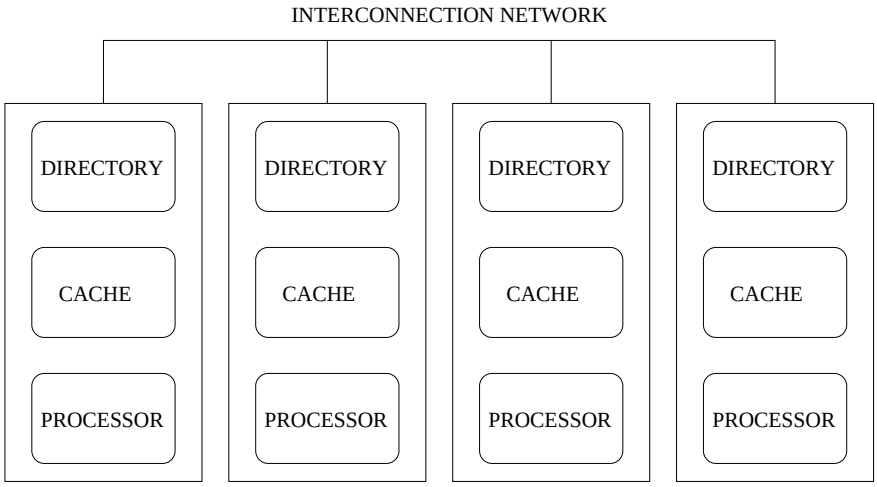
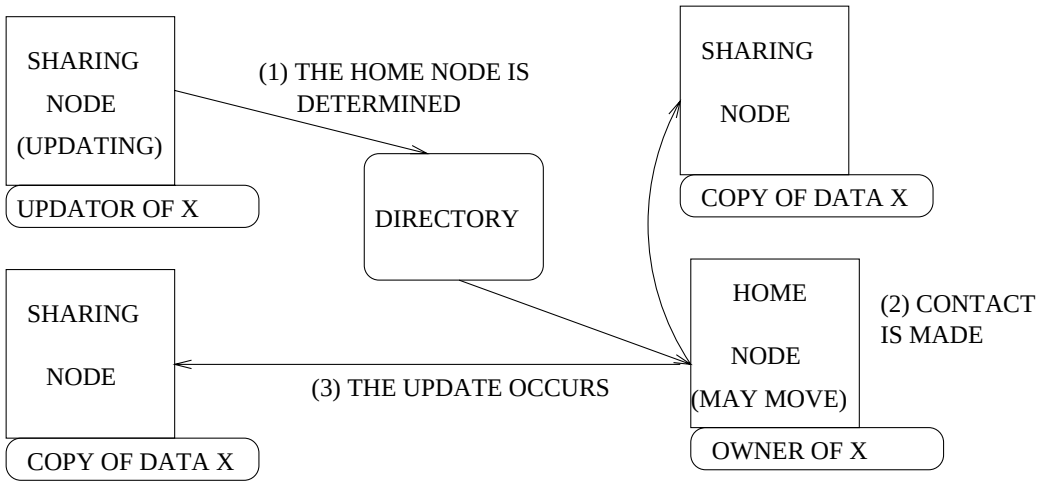


Figure 2.6. A Typical COMA Machine



NOTE: X represents a piece of data which must be updated. The HOME NODE owns X, but is not the node performing the update. The coherency protocol will determine the mechanism for in a semantically correct fashion. Repeated accesses to X may cause its home node to be moved to the accessing node. Additionally, a directory is required to determine the home node.

Figure 2.7. A COMA machine during an update.

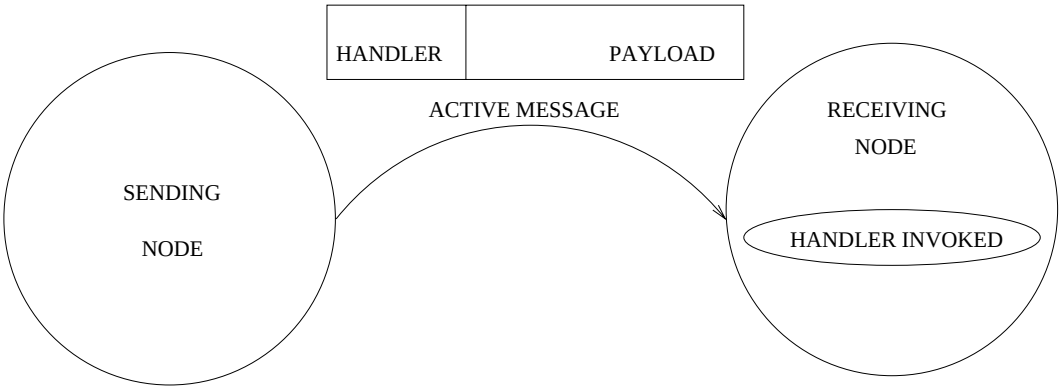


Figure 2.8. Active Messages

software including the Data Diffusion Machine (DDM) [37], the Princeton SHRIMP [11], and Simple COMA [32]. While the construction of the directory to allow the migration of data throughout the system is a complex issue, designers of COMA architectures focus tremendous amounts of time on the cache coherency protocols which allow them to compete with the speed of CC-NUMA machines. Unfortunately, it is this coherency protocol more than the directory structure which limit the scalability of COMA machines.

2.7 Active Messages and the J-Machine

Active Messages [36, 9] seek to maximize the efficiency of parallel machines by minimizing the overhead associated with communication and allowing for the masking of communication latency by overlapping communication and computation. Rather than a typical message passing system (such as MPI [16]) which provides a mechanism of communication between nodes, Active Messages further allows for the invocation of a *handler* upon reaching a given node. The purpose of this handler is two fold: first, it resides in user space, and, given the appropriate system configuration, can spare the application from an expensive call to the Kernel to handle the message; and second, since the handler is invoked upon receipt of the message,

systems do not sit idle waiting for communication to occur.

There are several similarities between Active Messages and Parcels. Clearly the notion of invoking some sort of handler (or in Parcel parlance "command") is the same. Parcels, however, are likely to provide increased flexibility in that the the invoking of a command does not require that the command be pre-resident on a given node, as is the case with a handler.

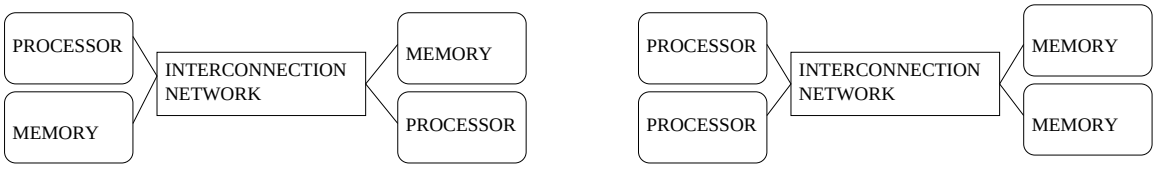
This is very similar to work done on the MIT J-Machine [10], which attempted to create an inexpensive massively parallel computer by supporting primitive mechanism for efficient communication, synchronization and naming of fine grain threads.

2.8 Active Pages

Active Pages [29] (developed at UC Davis) attempt to move beyond the von Neumann bottleneck by shifting data intensive computations away from the main processor towards simple PIMs which are constructed from reconfigurable logic. An Active Page is, therefore, the data and its associated functions. This greatly reduces the traffic between the processor and memory for these applications.

2.9 Tera MTA

The Tera Multi-Threaded Architecture (MTA) [2], which the successor of the Horizon [24, 35] is a highly parallel machine which seeks to achieve performance by emphasizing throughput and parallelism over speed and complexity. The architecture supports simultaneous multi-threaded (SMT), which allows for the interleaving of code from different threads. A *thread* is merely a sub-unit of execution with in a program which shares the same address space as any other thread. Since threads are largely independent of each other, they can generally be executed in any order provided that a sufficient mechanism for synchronization exists.



(a) A UNIFORMLY DISTRIBUTED CONFIGURATION

(b) A "DANCE HALL" CONFIGURATION

Figure 2.9. Types of Processor and Memory Distributions

Each Tera node allows for a zero cost context switch between threads, but is otherwise quite simple. The instruction set is quite simple, and optimized so as to eliminate the need for complex pipelining. Instruction encoding is said to be "horizontal" or moderately VLIW in that each cycle allows for the execution of one memory reference, one ALU operation, and an additional control or ALU operation. There is **no cache**, which eliminates the need for a coherency protocol. Because thread contexts can be switched at will, all code is executed in order. Rather than attempting an out of order or speculative execution, the system merely switches thread whenever there is any event which may cause latency. Most significantly, each node in the system is capable of supporting enough threads so that the latency of a memory accessed may be masked. In fact, the Tera hardware and compiler attempt to trade locality for parallelism whenever possible. All loads are done in the granularity of on 64-bit word, and memory addresses are distributed throughout the system to eliminate "hot-spots" (that is consecutive memory addresses **are not** next to each other physically). See Figure 2.9.

A thread context consists of one 64-bit stream status word which contains the thread's PC and current machine state, 32 64-bit General Purpose Registers, and 8 64-bit Target Registers used in branching. This means that each thread has 2624 bits of context are associated with each thread. There are a maximum of 128 thread contexts available on each node, which means that the register file must hold a total of 41 KB of data. This is similar to the size of a data cache on a typical workstation.

Table 2.1. Semantics of Tera's Full and Empty Bits

VALUE	Meaning on LOAD	Meaning on STORE
0	always read	write then set full
1	reserved	reserved
2	wait for full and leave full	wait for full and leave full
3	wait for full and set empty	wait for empty and set full

Note: "fullness" indicates data is located in memory and ready to be accessed while "emptiness" indicates that there is no data currently available.

The only synchronization mechanism provided for is through the use of memory. Two "full/empty" bits are provided for each 64-bit word. The semantics of the full/empty bits are described in Table 2.1.

The Tera Interconnection Network is a high speed 3d toroidal mesh. Tera is considered an SMP in that there are processing nodes and memory nodes connected to the network. Both nodes are intermingled to avoid the typical "dance-hall" configuration of most SMPs where processing and memory elements reside on opposite sides of the network. Again, even in the design of the Interconnection Network, parallelism is emphasized in the extreme.

The success of the Tera is completely dependent upon the ability of the programmer and compiler to provide sufficient parallelism (in the form of enough threads) to keep the system busy at all times. Since locality is of no importance, a large number of threads must be available to run on each node.

2.10 Relevance to PIM

It is well known that coherency protocols are expensive. Both CC-COMA and CC-NUMA architectures are not considered scalable beyond few thousand nodes. The Multi-Threaded advantages upon which the Tera MTA relies to achieve performance is indeed provocative since a simple synchronization mechanism can be substituted

for a full coherency protocol. Providing sufficient threads on a system of 1,000,000 or more nodes, however, may prove impossible. Multi-threading can still be used to mask tremendous amounts of latency and provide high node utilization.

Parcels further allow for many of the features of Active Messages. Rather than being constructed on top of commodity workstations, however, Parcels are integrated with the architecture and multi-threading.

CHAPTER 3

THE BENCHMARKS

> So assess them to find out their plans, both the successful ones and the failures. Incite them to action in order to find out the patterns of their movement and rest.
>
> – Sun Tzu, *The Art of War*

The benchmarks employed in this work are primarily designed to be difficult for the memory system to handle. It has long been known that large data sets which are accessed non-contiguously are the bane of efficient high performance computing. Because these applications are typically unable to take advantage of memory-access optimizations, they perform approximately two orders of magnitude below peak rates [4]. However, this class of applications, which will be hence forth referred to as *data intensive applications*, typically form the core of scientific and engineering computation. Examples include high-definition imaging, relational and object-oriented databases, and circuit simulation. Clearly understanding how these applications behave on PIM systems is of paramount importance.

This chapter will describe the benchmarks used throughout this work and explain how they exhibit data intensive behavior. Additionally, it will describe some of the fundamental properties of the benchmarks, including how memory accesses are distributed among the code, data, and stack segments. Additionally, the use of a uniquely PIM optimization will be discussed: using the basic memory constructs (such as row buffers) as a simple caching mechanism.

3.1 Benchmarks Under Consideration

The initial research for this work used more traditional benchmarks such as the SPEC95 Suite [28]. However, these benchmarks do not typically exhibit the characteristics of data intensive applications. Quite the opposite, the data sets chosen for these applications have been designed to be quickly captured in the data cache of a typical workstation so as to emphasize raw computing performance. While they provided some initial understanding of the overall structure for memory systems, as well as valuable (independent) validation for many of the simulators written, the data they produced was largely uninteresting when cast in the light of the data intensive applications which PIM systems wish to exploit. Thus, beyond their use as an initial learning tool and a validation system, the overall application set used in this thesis does not reference any SPEC95 programs.

The next benchmark to be examined was based on a simple database implementation (by Pedro Diniz at USC's Information Sciences Institute) servicing a set of object-oriented data base queries defined in the *oo7* benchmark standard [8]. This provided valuable information into the nature of data-intensive applications, as well as an additional opportunity to validate the results of simulation. There were primarily two difficulties in using this benchmark: first, the back-end implementation was somewhat simplistic and unrealistic; and second, the queries are designed for accessing a CAD document system, and therefore require large amounts of data to be returned upon completion. While this tests the transfer rate of large blocks in a database, it exhibits streaming behavior rather than data intensive behavior, and is therefore less interesting for the purposes of this work.

With the completion of the Data Intensive Systems (DIS) Benchmark Suite [4], an ideal set of appropriate applications became available. Since the DIS Suite contained a cleaner object oriented database implementation, *oo7* was used only for

validation purposes.

One additional benchmark was added, a simple *Molecular Dynamics* simulation which is in essence an implementation of the classic N-Body problem using a leapfrog integrator [19]. This type of simulation is at the core of solving the protein folding problem, which IBM and others are investigating with tremendous vigor.

3.2 DIS Data Management

The Data Management Benchmark [4] implements a simplified object-oriented database with an R-Tree indexing scheme [17, 23]. R-Trees have the following properties:

- They are height balanced (ie, all leaves are at the same level);
- If M is the order of the tree, and k is a constant, every node has between kM and M index entries (except the root);
- A sub tree may contain a hypercube index if that hypercube can cover the entire subtree;
- The root has at least two children unless it is a leaf.

The system responds to three operations: *insert*, *query*, and *delete*. Queries can be either key based or content based. All Data Management runs consist of all three operations (building the tree, querying it, then deleting it), however, only the query operation is examined during simulation. In this case, an index of approximately 9 MB is created.

The query function is given an R-Tree, a search key, and some non-search key attributes. It returns all records which are consistent with the search key and the non-key attributes.

3.3 DIS FFT

The Multidimensional Fourier Transform [4] is extensively used in technical fields, including image processing and digital signal filtering. In fact, a Fast Fourier Transform (FFT) (from the same code based) occurs in both the Ray Tracing and Method of Moments benchmarks to be described later. The benchmark implements a 3-dimensional Fourier Transform using an FFT [13] algorithm. It is believed that the results of a three dimensional system will demonstrate architectural properties indicative of higher dimensional operations. The input matrices for this benchmark are approximately 45 MB in total size.

A Discrete Fourier Transform (DFT) in 3 dimensions is defined by the following equation:

$$F(x,y,z) = \sum_{k_3=0}^{Z} \sum_{k_2=0}^{Y} \sum_{k_1=0}^{X} e^{\frac{2\pi i k_3 z}{Z}} e^{\frac{2\pi i k_2 y}{Y}} e^{\frac{2\pi i k_1 x}{X}} f(k_1,k_2,k_3) \tag{3.1}$$

where f is the complex three-dimensional array of size X x Y x Z and F is the output Fourier transform of f.

The benchmark uses the FFTW library to choose an appropriately optimized FFT to solve the DFT given above based on the input data.

3.4 DIS Method of Moments

Method of Moments algorithms are frequency domain techniques for computing electro-magnetic scattering from complex objects (see Figure 3.1). Typical implementations employ direct linear solvers, which are of high computational complexity. Consequently, these algorithms can only be reasonably applied to low frequency problems. Current research has produced faster solvers, however, applying them to high frequency problems is limited by the speed of main memory because data reuse is quite low and memory accesses exhibit non-unit stride.

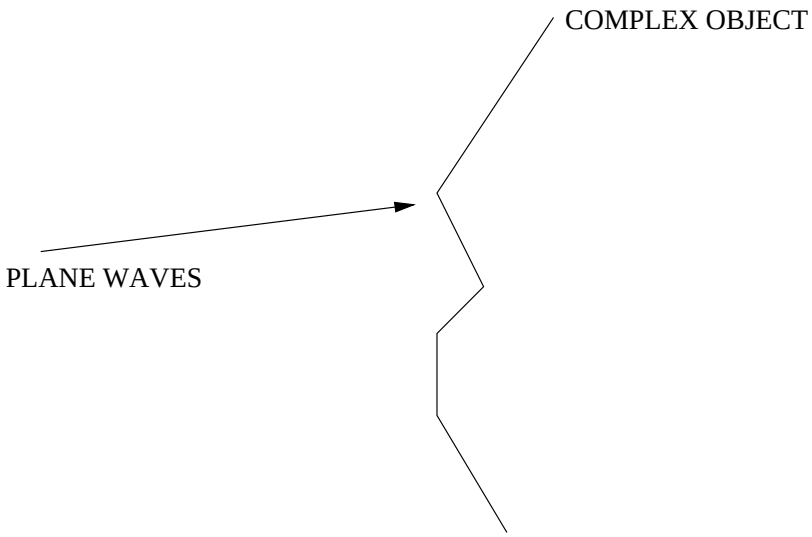


Figure 3.1. EM Scattering

The DIS Method of Moments code comes from the Boeing implementation of a fast iterative linear solver for the Helmholtz equation [12, 15, 14].

The input set requires the creation of several complex data structures constituting approximately 530 MB of memory.

3.5 DIS Image Understanding

Image Understanding attempts to detect and classify objects within an image. The implementation requires three phases: first, morphological filtering, in which a spatial filter is created and applied to remove background clutter; second, determination of the region of interest; and third, feature extraction. In this case, the benchmark examines a 9MB image for a set of pre-computed objects. See Figure 3.2.

3.6 DIS Simulated SAR Ray Tracing

The simulation of Synthetic Aperture Radar is divided into three parts:

- Geometry Sampling

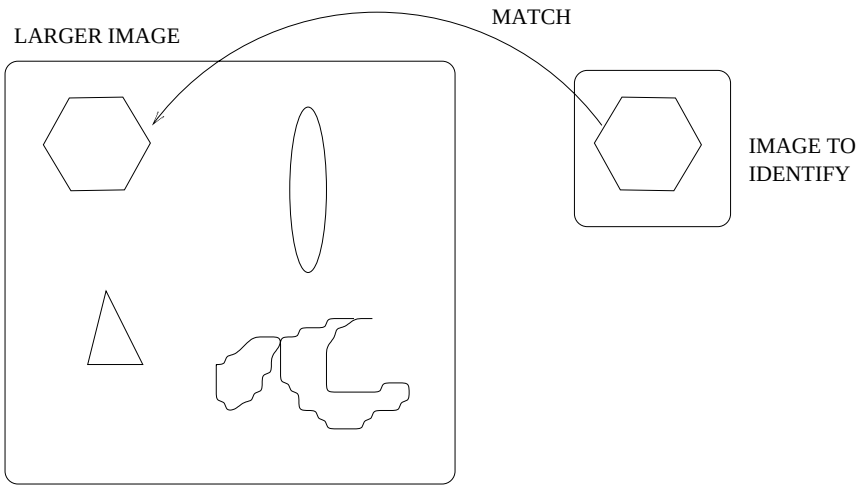


Figure 3.2. Image Understanding

- Electro-magnetic Scattering Prediction
- Image Formation

Of these, the geometric sampling (which is the actual ray tracing) was chosen for analysis in this work. While the Image Formation step is also quite computationally intensive, the geometric sampling step represents a more general problem and classically exhibits the data intensive behavior under examination. In general, the problem consists of sending rays out from a fixed point and determining where they intersect with other objects. See Figure 3.3.

This run considered ray tracing over an 8 MB image representative of a Simulated SAR problem.

3.7 Molecular Dynamics Simulation

The molecular dynamics simulation is primarily an integration of the equations of Newtonian Mechanics over a force field. In this particular case, the simulation is of argon atoms in three dimensions. Since the argon is an inert gas, there are no bonded forces to be computed (which is less relevant as they are much simpler to compute

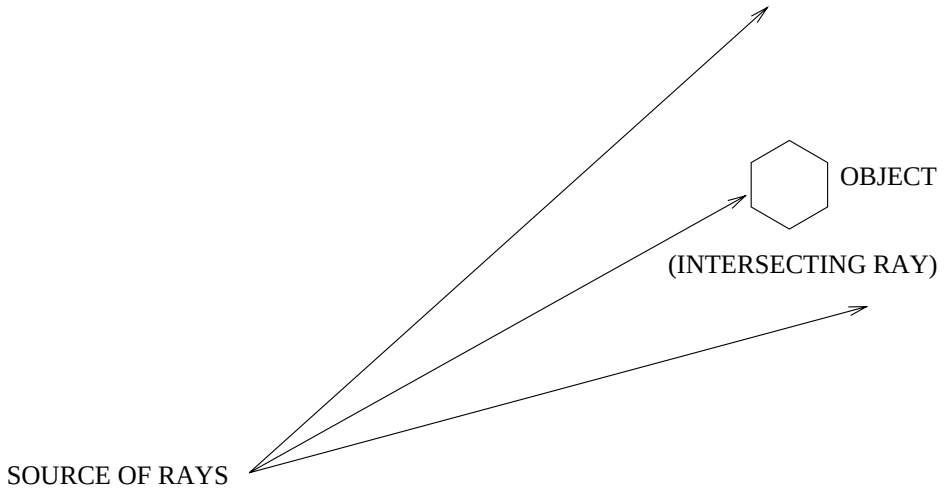


Figure 3.3. Ray Tracing

than the non-bonded forces, and exhibit higher degrees of locality). Furthermore, there are no electrostatic forces, making the only non-bonded calculation van der Waals forces. The simulation is significantly more simple than most implementations. However, the key computations, including a non-bonded force calculation during the integration, are well exhibited.

The system modeled consists of 125000 atoms at 35 Atmospheres. The data sets are broken up into a 3-d cube of boxes, each of which interacts with its six neighbors. The data set size for the primary matrices used during the computation is approximately 14 MB. Each box contains at most 8 atoms. See Figure 3.4.

Simulation for this benchmark occurs only during the non-bonded force computation.

3.8 General Memory Access Characteristics

Table 3.1 indicates how many data or stack reads and writes are present for each benchmark on a per instruction basis. This is equivalent to the probability that for a given instruction fetch a read or write to the stack or data segments will be made.

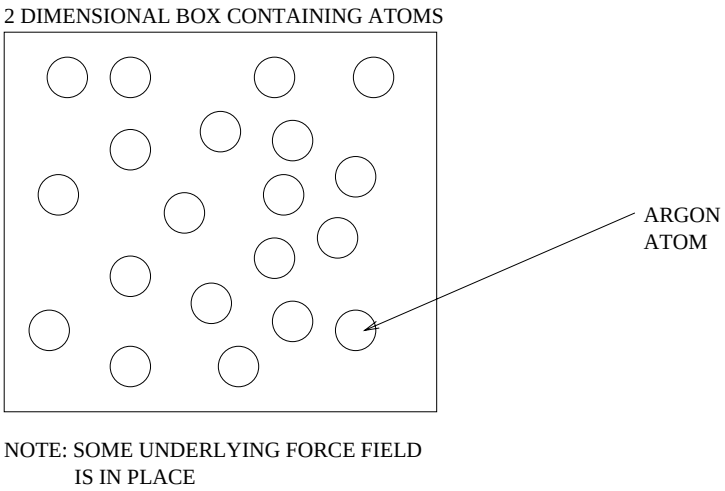


Figure 3.4. Molecular Dynamics Simulation

Aside from simply providing information about the instruction mix and likelihood of various types of reads or writes, the table indicates the degree of data intensive behavior in that it shows how data much is required to support the computations.

The complete instruction frequencies, computed by Shade's *ifreq* program, for each benchmark can be found in Appendix A.

The memory references frequencies which follow were computed with a Shade simulation that tracks all memory references a determine if they access the code segment (during an instruction fetch), or the data or stack segments (during a load or store). The stack segment is determined by examining the contents of the stack and frame pointers.

3.9 Row Buffer Re-usage

The PIM memory system (described in Section 2.1.1) has multiple latencies that are incur ed for local memory accesses. If a local access hits the currently open wide word, there is zero latency to read perform the read (because the wide word is the basic unit of memory access for the PIM). Furthermore, if a given memory access

Table 3.1. Benchmark Read and Write per Instruction Statistics

PROGRAM	DATA R/I	DATA W/I	STACK R/I	STACK W/I
DIS DM	0.0885	0.0263	0.1643	0.0712
DIS FFT	0.1248	0.0757	0.0413	0.0387
DIS MoM	0.0793	0.0173	0.2248	0.0548
DIS IU	0.3478	0.0931	$8 * 10^{-9}$	0
DIS RAY	0.0389	0.0001	0.2994	0.1276
MD	0.1406	0.0230	0.1749	0.0201
AVERAGE	0.1367	0.0392	0.1508	0.0521

R/I = Reads per instruction executed
W/I = Writes per instruction executed

Table 3.2. Open Row Buffer Cache Hit Rate

BENCHMARK	1	2	4	8
DIS DM	35.2663 %	49.4695 %	66.3240 %	88.4979
DIS RAY	58.8541 %	86.9943 %	87.0956 %	87.1533
DIS IU	22.8428 %	99.6614 %	99.6614 %	99.6614
DIS FFT	33.5064 %	35.5359 %	96.1673 %	97.9683
DIS MoM	54.3057 %	78.1146 %	80.4608 %	88.1132
MD	61.7237 %	83.9452 %	91.1372 %	97.4895

is located in the open row register, only a single cycle is necessary to perform the read (to transfer the data requested to the processor's register file). It is only when the open row register is missed that a full memory access must be performed.

This section asks the question: what if there were several wide words or open row registers that were used as a cache for data accesses? (Ignoring instruction fetches and stack accesses.) A Shade simulation was performed to answer that very question. Every data reference was traced and compared against up to 8 open row buffer registers (holding 2 K-bits of data each). The same references were compared to up to 8 wide words (which can hold 256 bits of data each).

Table 3.2 shows the hit rate for the "cache" of up to 8 open row registers.

Table 3.3. Wide Word Cache Hit Rate

BENCHMARK	**1**	**2**	**4**	**8**
DIS DM	32.6029 %	45.3322 %	57.5982 %	77.0256
DIS RAY	41.1372 %	54.2241 %	74.3265 %	74.3538
DIS IU	22.6443 %	97.0264 %	97.0264 %	97.0264
DIS FFT	31.6672 %	32.4657 %	88.8958 %	90.5215
DIS FMM	48.4598 %	65.0872 %	65.7167 %	66.9643
MD	47.6915 %	64.0381 %	71.8059 %	75.8537

Table 3.3 summarizes the wide word hit rate for up to 8 wide words.

These table show that each of the benchmarks shows a high degree of reuse among both open rows and wide words. In fact, 8 open rows (which is only 2 KB of data) showed a hit rate of over 87% for every benchmark. Similarly, 8 wide words (a mere 256 bytes) achieved a reuse rate of over 66% for every benchmark, and in some cases well into the 90% range.

3.10 Conclusions

Although these benchmarks are data intensive in the sense that they exhibit low reuse between large data structures, there is a high degree of internal spatial locality. The reuse of row buffers (or, similarly, adding a small cache between the memory macro and the processor) is extremely promising and could be achieved at very low cost. Since there must be at least one row buffer, there is the potential to cut the cost of $\frac{1}{3}$ of memory accesses to only one clock cycle at zero cost.

CHAPTER 4

SPATIAL OVERHEAD

> Economic action is primarily oriented to the problem of choosing the end to which a thing shall be applied; technology, to the problem, given the end, of choosing the appropriate *means*.
>
> – Max Weber, *Economy and Society, Volume I*

The experiments in this work evaluate the performance of potential PIM memory system implementations. During the course of various experiments, the PIM memory macro will be treated as a cache or as paged memory. This chapter quantifies the overhead involved with these uses. The measure of overhead, in this case, is the extra bits needed to construct and manage a cache or paged memory. In the case of a cache, these bits are contained in the tag and other "extra" information (such as age) which must be stored with the data value. In the case of a paged space, the overhead is measured by the size of the page table needed to translate addresses.

4.1 Address Space Assumptions

For the purposes of this chapter, the address space is assumed to be 32-bits. This allows for 4 GB worth of data to be uniquely addressed. This is the standard size for an address space on a modern workstation, but is relatively small when considering the construction of a 1,000,000 node PIM array which may be capable of addressing a Tera-byte of physical memory or more. However, this accurately represents the construction of small to medium size PIM systems, and the appropriate scaling

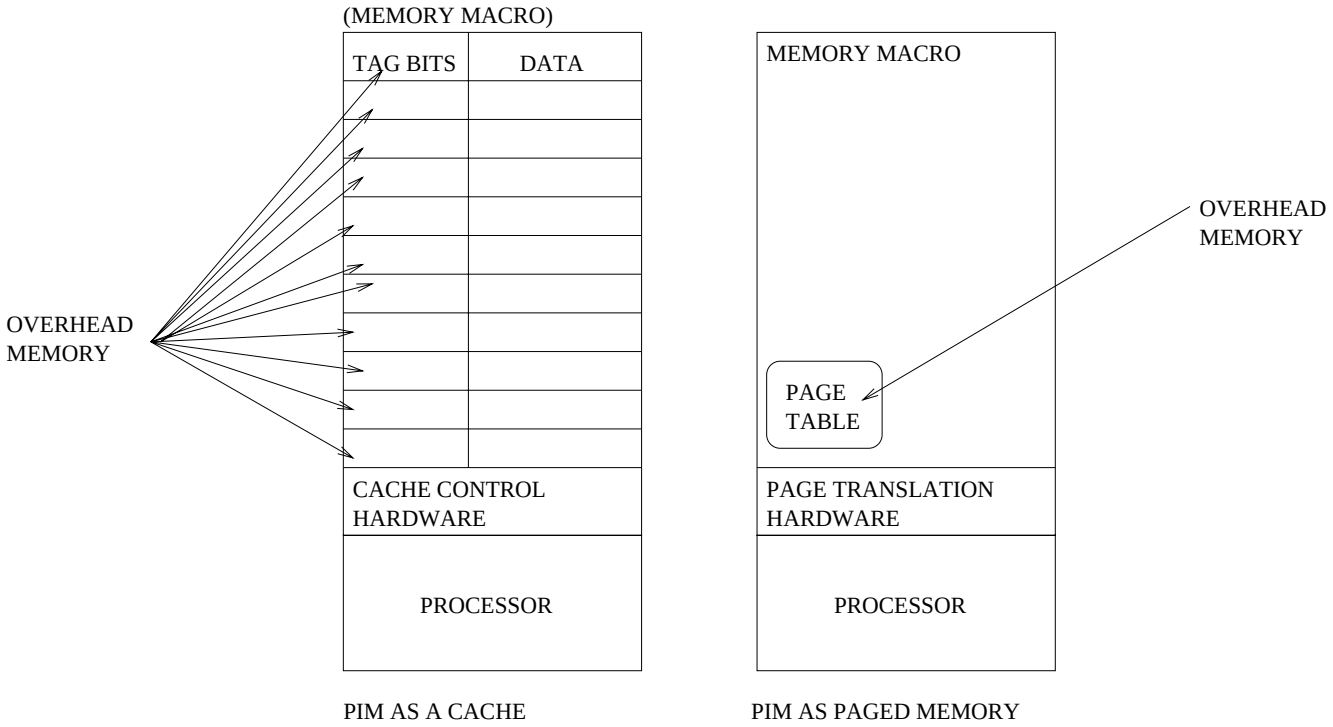


Figure 4.1. PIM as a Cache and as Paged Memory

can be done for large machines. Figure 4.1 shows the organization of these two configurations and where the overhead bits are stored.

4.2 "Centralized" Page Space Overhead

One of the most common methods of translating a name into a physical location is the use of a *page table*. The "name" is used to index into the table which yields the location of that object. PIMs, when combined together, may use such a table to determine where objects reside both on the node and off the node. This section will describe the overhead of using a page table to perform address translation. Assumptions regarding the page table size are minimal. It is assumed that the entries in the table contain only a validity bit (to determine if the page exists) and a physical address. It is further assumed that the page table could reside on a single PIM or be distributed across multiple PIMs. Additionally, a PIM may wish to hold

a very small page table locally in order to translate local addresses.

4.2.1 Directory Based Page Table Overhead

Table 4.1 shows the total amount of memory required (system wide) to store page tables with pages of various sizes. The overhead number represents the size of a singe copy of the entire page table for a given program (this could be distributed among many PIMs, or if space allows it could reside on a single PIM). It can easily be seen that a 4 KB page size is too large to fit within a single node of the smallest (2 MB) configuration.

The table further shows that a page size of 256 KB, which will be shown to be reasonable in Chapter 5, requires only a 31 KB page table (about 1.57% of the smallest proposed PIMs memory space). This small overhead could potentially allow a PIM node very fast off chip translation. Furthermore, a page table containing entries the size of the proposed PIMs, particularly as the PIMs get large, contains so few entries that all the translations could be performed in hardware. These large page sizes will be shown to be reasonable in Chapters 6 and 7.

4.2.2 Single Node Page Table Overhead

Perhaps a more interesting measure of page table overhead can be seen in Table 4.2 which shows the number of entries required *locally* for a given node to translate just its own addresses. Since PIM optimizations are centered around the fast local accesses and high local bandwidth available on chip, it is very important that addresses which access the local memory macro be translated very quickly. If one were to do such a translation using a TLB to identify local addresses (see Figure 4.2), Table 4.2 represents exactly the number of entries which the TLB would have to contain to cover the entire memory macro's address space.

Table 4.1. Total Page Table Overhead

Page Size	**# of Entries**	**Entry Size**	**Total Overhead**	% of 2 MB PIM
4 KB	1,048,576	21 bits	2,752,512 bytes	131.25
8 KB	524,288	20 bits	1,310,720 bytes	62.5
16 KB	262,144	19 bits	622,592 bytes	29.69
32 KB	131,072	18 bits	294,912 bytes	14.06
64 KB	65,536	17 bits	139,264 bytes	6.64
128 KB	32,768	16 bits	65,536 bytes	3.125
256 KB	16,384	15 bits	30,720 bytes	1.47
512 KB	8,192	14 bits	14,336 bytes	0.68
1 MB	4,096	13 bits	6,656 bytes	0.32
2 MB	2,048	12 bits	3,072 bytes	0.15
4 MB	1,024	11 bits	1,408 bytes	0.067
8 MB	512	10 bits	640 bytes	0.0305
16 MB	256	9 bits	288 bytes	0.0137
32 MB	128	8 bits	128 bytes	0.0061

Table 4.2. Local Page Table Overhead (in number of entries)

PAGE SIZE	**2 MB PIM**	**4 MB**	**8 MB**	**16 MB**	**32 MB**
4 KB	512	1,024	2,048	4,096	8,192
8 KB	256	512	1,024	2,048	4,096
16 KB	128	256	512	1,024	2,048
32 KB	64	128	256	512	1,024
64 KB	32	64	128	256	512
128 KB	16	32	64	128	256
256 KB	8	16	32	64	128
512 KB	4	8	16	32	64
1 MB	2	4	8	16	32
2 MB	1	2	4	8	16
4 MB	-	1	2	4	8
8 MB	-	-	1	2	4
16 MB	-	-	-	1	2
32 MB	-	-	-	-	1

4.3 Cache Overhead

The overhead presented in this section assumes that the PIM memory macro is used to arbitrarily cache data (similar to a CC-NUMA configuration). Caches must store

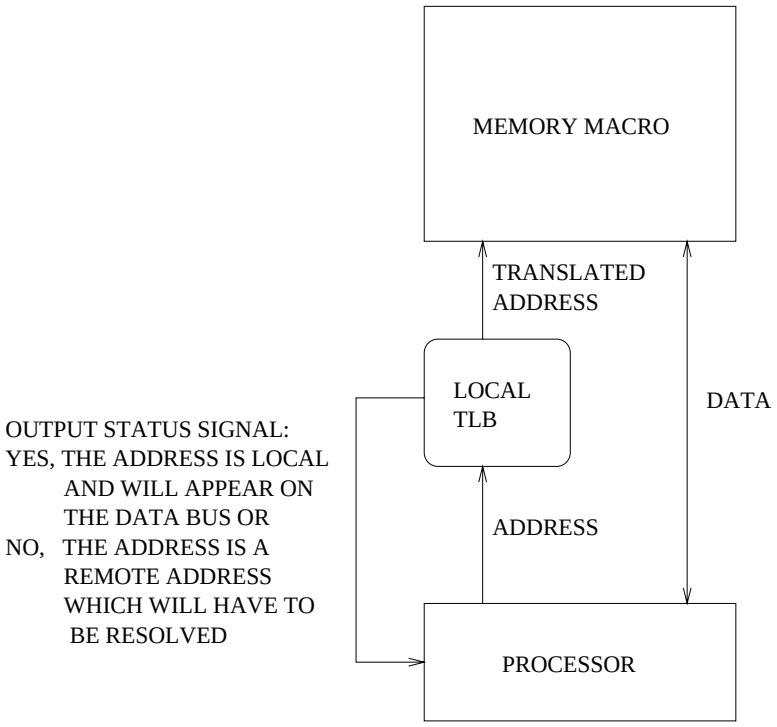


Figure 4.2. PIM Equipped with a Local TLB

some number of tag bits (as well as other information such as age and validity bits) for each data item stored in the cache. This overhead information is used to identify the data and its state. The information which follows represents the size of these overhead bits for various PIM sizes.

Four cache configurations are considered: a direct mapped cache with a 256 bit and 2 K-bit data size, as well as a 4-way and 8-way set associate cache, both with 256 bit data sizes. These numbers correspond directly to the size of a wide word, and a full row. Each of the cache configurations requires a validity bit and a tag. The set associative caches, which perform eviction based on age, each require 2 additional bits of aging information. These and the validity bits will be considered "overhead" bits and are required for each word. The tag bits are accounted for separately.

It is assumed, that regardless of the PIM size, a 32-bit address must be reconstructed from the tag bits and the index. The overhead presented in this section is

on a per node basis – to scale it to the entire memory space, it must be multiplied by the number of nodes in that space.

4.3.1 256 bit Direct Mapped

The 256 bit direct mapped cache, while obviously requiring more overhead than any of the page space configurations, still only requires between 9.38% and 10.94% additional memory on each node depending on the size.

Table 4.3. 256 Direct Mapped Cache Overhead

Cache Data Size	**Overhead Bits**	**Tag Bits**	**# of Entries**	**Total Overhead**
2 MB	1	27	64 K	229,376 bytes
4 MB	1	26	128 K	442,368 bytes
8 MB	1	25	256 K	851,968 bytes
16 MB	1	24	512 K	1,638,400 bytes
32 MB	1	23	1 M	3,145,728 bytes

4.3.2 2 K-bit Direct Mapped

The 2 K-bit direct mapped cache is simply a constant factor improvement over the 256 K-bit direct mapped cache presented above.

Table 4.4. 2 K-bit Direct Mapped Cache Overhead

Cache (Data) Size	**Overhead Bits**	**Tag Bits**	**# of Entries**	**Total Overhead**
2 MB	1	24	8 K	24,576 bytes
4 MB	1	23	16 K	47,104 bytes
8 MB	1	22	32 K	90,112 bytes
16 MB	1	21	64 K	172,032 bytes
32 MB	1	20	128 K	327,680 bytes

4.3.3 4-way Set Associative

The 4-way set associative cache requires approximately 1.6% to 2.1% overhead.

Table 4.5. 4-way Set Associative Cache Overhead

Data Size	**Overhead Bits**	**Tag Bits**	**# of Entries**	**Total Overhead**
2 MB	3	18	16K	43,008 bytes
4 MB	3	17	32K	81,920 bytes
8 MB	3	16	64K	155,648 bytes
16 MB	3	15	128K	294,912 bytes
32 MB	3	14	256K	557,056 bytes

4.3.4 8-way Set Associative

Again, a constant factor increase in overhead is provided by increasing from 4-way to 8-way set associative. In this case, between 0.09% and 1.1% overhead is possible.

Table 4.6. 8-way Set Associative Cache Overhead

Cache (Data) Size	**Overhead Bitsc**	**Tag Bits**	**# of Entries**	**Total Overhead**
2 MB	3	19	8 K	22,528 bytes
4 MB	3	18	16 K	43,008 bytes
8 MB	3	17	32 K	81,920 bytes
16 MB	3	16	64 K	155,648 bytes
32 MB	3	15	128 K	294,912 bytes

4.4 Conclusions

Although these calculations are relatively simple, and the outcomes are not particularly shocking, it is very important to keep in mind the overhead involved with configuring a PIM to be either an area for paged data or a cache. It is particularly important to realize the impact on the local PIM node (ie, in storing just its local

page translations, which must be very fast, or in storing the overhead bits required by a cache).

Page table configurations are obviously add significantly less overhead than do caches (by 1 to 2 orders of magnitude)

CHAPTER 5

WORKING SET CRITICAL MASS

> Sometimes a man seeks what he hath lost; and from that place, and time, wherein he misses it, his mind runs back, from place to place, and time to time, to find where, and when he had it; that is to say, to find some certain, and limited time and place in which to begin a method of seeking. And from thence, his thoughts run over the same places and times, to find what action, or other occasion might make him loose it. We call it Remembrance, or Calling to mind; the Latins call it *Reminiscentia*, as if it were a *Re-conning* of our former actions.
>
> – *Leviathan*, Chapter II, Thomas Hobbes

The goal of a PIM based system is to circumvent the von Neumann bottleneck by merging logic and memory onto a single die and exploiting the available bandwidth therein. The limited resources of any near term PIM node make it impossible for a single node to capture the entire data demands of a large problem, however, such a node must be able to capture a meaningful subset, which will be termed a *working set*. Thus, this chapter will quantify to what extent PIMs of various sizes can capture a valid working set.

A typical microprocessor captures a working set through the use of a cache. Because a PIM is capable of supporting significantly more on-chip memory than a standard microprocessor, a cache is not the only possible mechanism for capturing a working set. A PIM can also use a *page space*, which is more typical of how a workstation manages out of core storage. (Figure 5.1 shows a working set and both possible representations.) The benchmarks under examination (described in Chapter 3) were chosen specifically because they represent some of the most difficult

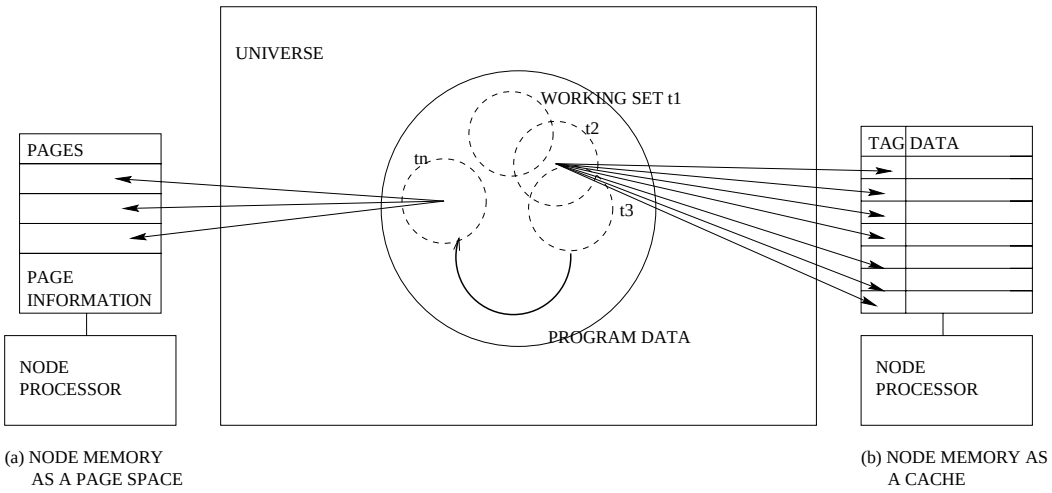


Figure 5.1. The Working Set and Its Time Evolution

for any memory system to handle. They exhibit an irregular stride through main memory (in the form of large sparse matrix operations or pointer chasing) and have relatively low reuse. Therefore, the choice of method to use in capturing a working set (ie, a page space or cache) is unclear.

This chapter will show that a page space is a far superior configuration to a cache, and that, in fact, caches are largely ineffective for very large local memory sizes. This will be done by defining a new metric for analysis, the Cumulative Instruction Probability Density (CIPD), and examining the results of both cache and page experiments in that light. Because the CIPD provides too much information to be easily assimilated, the standard Miss Rate metric will also be used to eliminate trivial cases. The experiments will also show the best configurations for both a large cache and page space. Most importantly, it will be demonstrated that a 2 MB PIM can sufficiently capture a meaningful working set, as well as which configurations above 2 MB yield significant improvement.

5.1 Experimentation

For the purposes of the experimentation the local memory macro size is assumed to be no greater than 32 MB (which represents a very large upper-bound, considering the size of devices which could currently be implemented is somewhere between 2 and 8 MB). The "success" of the PIM capturing a valid working set is measured in terms of instruction run lengths between misses to data in the working set; specifically, the longer the machine can run while hitting only local data, the more successful that configuration is. A *run length* is defined as the number of consecutive instructions which are executed before the contents of the working set are altered by a request for data not already in the set. In other words, given some subset of data from the the entire program, the run length represents the length of time (in terms of actual time or instructions executed) that the program can execute before data from outside the working set is requested. Additionally, miss rates are presented as an aggregate over the entire program. See section 5.2 for further information on the measurement of miss rates and run lengths.

Since the working set which can be contained on one node is too small to represent the entire program, a mechanism for fetching non-local data and updating the working set must be defined. In a traditional cache, this is represented by a replacement policy. When a data request is made to the cache which cannot be serviced, a *miss* is said to occur. The cache then requests the appropriate piece of data from main memory. When the requested data arrives, if the cache is full, some element within the cache must be ejected so that the new data can enter. The decision as to which element is ejected is termed the cache's *replacement policy*. In all experiments in this work, the replacement policy is true Least Recently Used (LRU) [31].

5.1.1 Experimental Configurations

In order to simplify the analysis, the experiments are split into two primary configurations: first, a system where the memory macro acts as a *page space*, which is defined as an area in memory capable of accepting relatively large contiguous chunks of memory (4K or greater, in this case), and where the replacement is managed by software; or, second, as a traditional cache, where relatively small contiguous chunks of memory (2 K-bits or less) have placement managed by the hardware. Information regarding the overhead of each of these schemes is discussed in detail in Chapter 4. In either configuration, it is assumed that the size of the memory is the size of the data to be stored and does not include any potential overhead information (such as the tag bits or page tables).

5.1.2 Additional Validation

Since the page space form can be viewed as a large sector cache (and conversely, the cache can be viewed as a page space with extremely small pages), the fact that simulators were developed to implement both in different ways allows one simulation to be used to validate the results of the other. The page space simulation keeps a sorted list of accesses with the most recently used in the front, while the cache simulation maintains a traditional cache structure with tag bits and all the appropriate cache data. Since the results of either experiment, when run on the same program with equivalent configurations, should be the same, the two simulators were used to validate each other during the initial stages of development using the SPEC95 Integer Benchmark Suite [28] as well as small configurations from each of the DIS benchmarks [4] and oo7 [8]. Further validation was performed using the cache simulation which is included with the Shade [34] package (*cachesim5*), which is capable of handling only small cache configurations.

5.1.3 Assumptions

The experiment assumes a "cold" machine upon startup, which means that no working set has been pre-loaded and the first load or store encountered will generate a miss. This serves to show the start-up cost of loading the initial working set onto the node. Throughout the execution of the program, the shade based simulator tracks all memory references (loads, stores, and instruction fetches), and determines, on the basis of the appropriate address, which page or cache line is being accessed. Instruction fetches and data references are easily determined through examining the opcode of the instruction being executed, however stack references must be more carefully analyzed. By examining the "growth" of the stack (as measured by changes in the stack and frame pointers) from the top location of memory down, the simulator tracks all adjustments to the stack or frame pointer and assures the simulations understanding of the stack, its size, and its location is the same as that which the program generates. Examination of page usage histograms confirms this analysis to be correct as there are large unused sections of memory between the end of the heap and the beginning of the stack in each program.

Each of the page configurations examines all three types of accesses: instruction, data memory, and stack, and shows in each case that a relatively small working set captures all code and stack references (less than 16 KB, for example, nearly capturing all the necessary pages). Code and stack references will be examined in greater detail in Chapter 7. Since the working set for the data is demonstrably larger than that for the code or stack, the cache configurations examine only data references.

Each of the page and cache configurations were examined for memory macros sizes of 1, 2, 4, 8, 16, and 32 MB. The page space experiments examined pages of traditional sizes (4K, 8K), as well as significantly larger sizes (16K-256K). Not

surprisingly, the benefits of added spatial locality quickly became apparent for the larger page sizes.

The cache configurations are far simpler. Since the size of an open row in the memory macro is assumed to be 2 K-bits, and the size of a *wide word* (or that which is selected from the column decoders) on that same macro is 256 bits, four configurations were chosen: a 256 bit and 2 K-bit block sized Direct Mapped cache, representing potentially the simplest hardware configurations; a 256 bit block sizes 4-way set associative and 256 bit block size 8-way set associative cache. The 4-way consists of half an open row and the 8-way an entire open row. Somewhat counter-intuitively, the direct mapped caches tend to outperform the associative caches for large local memory sizes. This is because small caches require a greater amount of associativity to achieve high performance, while larger caches can rely upon the spatial locality generated from consuming large cache lines.

5.2 Metrics

There are primarily two metrics which will be presented throughout the rest of this chapter. The first, and simplest to understand, is the *miss rate*. It is, quite simply, the fraction of accesses which cause a miss over the number of accesses during the entire program execution. If A represents the total number of accesses and M represents the total number of misses, the miss rate is merely $\frac{M}{A}$. This is the traditional metric presented when examining the "efficiency" of caches.

However, since the measure of efficiency for the purposes of these experiments is run length between misses, the more detailed *Cumulative Instruction Probability Density*, or CIPD, is also presented. The CIPD is computed by dividing a program's execution up into streams of instructions for which no miss is generated, given the memory state of the machine at the first instruction in each stream. That is, the

first instruction encountered which generates a miss constitutes the beginning of the next stream, which means that the previous instruction is the end of the preceding stream.

Streams of the same length (in terms of number of instructions) are placed into buckets. The probability that a randomly selected instruction stream will be from a given bucket is then computed. If the CIPD is represented by the function $\Psi(L)$ where L is an instruction length, $\Psi(L)$ will return the probability that an instruction stream of length greater than or equal to L will be encountered. Thus, for any program, $\Psi(0) = 1$, and if γ represents the maximum length of any instruction stream, $\Psi(\gamma + 1) = 0$. Each of the CIPD graphs which follow represent exactly the function $\Psi(L)$ for each experiment. Ψ can also be used to determine the probability that an instruction stream of length less than or equal to L will be generated. This function, called $\Psi^*(L) = 1 - \Psi(L)$.

It should be noted that the graphs are constructed from individual data points determined during program execution. Since the Ψ always begins at 1 and eventually decays to 0, anything to the left of the beginning of the graph (usually 10^3 instructions) will rapidly reach 1. Similarly, the "end-points" presented are not the true end-points (since they should always become 0); rather they represent the probabilities of the largest instruction streams encountered (or in the case of miss rates, the cold-start cost). Rather than presenting the entire function, these starting and ending points were chosen to better represent the graph and include more information.

There is no notion of weight contained within the CIPD, which can be thought of as "time spent executing." Instruction streams of very long length will show a relatively low CIPD, but could potentially represent the most significant percentage of the overall execution time.

5.3 Interpreting the Results

Because the space explored during the course of experimentation was so large, giving a detailed discussion of each individual experiment would prove prohibitively long. Therefore, the results presented in this chapter center around the highlights of experimentation. The complete results for each experiment run in the formulation of this chapter can be found in Appendix B.

5.3.1 Miss Rates

As the introduction emphasized, the success of any configuration is measured in terms of CIPD. Since the CIPD is not easily summarized only the unique results are presented. For completeness, each experiment will be summarized with miss rate information. The reader may correctly assume that lower miss rates generally correspond to longer run lengths between misses.

5.4 Cache Results

Generally, the cache configurations proved less effective than did the page configurations. In terms of run lengths, they typically fared 1 to 2 orders of magnitude worse. This demonstrates that all of the benchmarks exhibited significant spatial locality. This fact is further reinforced by the fact that none of the set associative caches ever yielded the best configuration.

5.4.1 DIS Data Management

Figure 5.2 shows that 2 K-bit cache configuration proved far superior to any other. This is confirmed by the CIPD data in Appendix B. Additionally, increasing the size of the PIM helped very little using this configuration, although the set associative caches improved a great deal (indicating that the sets were often unfilled). The only

potential improvement is moving from a 1 MB to 4 MB PIM, however, the difference in miss rate (and corresponding CIPD) is negligible.

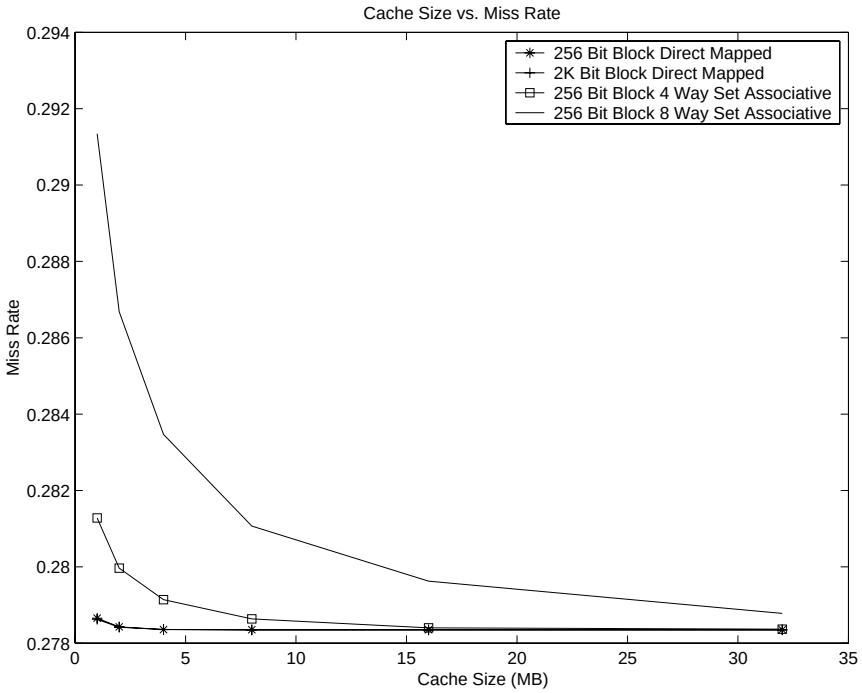


Figure 5.2. DIS Data Management Cache Size vs. Miss Rate

5.4.2 DIS FFT

The FFT benchmark strongly favored the 2 K-bit cache configuration. Neither of the set associative configurations came close to matching the direct mapped performance. The 256 bit direct mapped cache achieved the same performance as the 2 K-bit cache once the PIM size was 16 MB or greater, however the performance for the 2 K-bit direct mapped cache generally did not benefit from increasing the size of the PIM. (See Figure 5.3 for further details.)

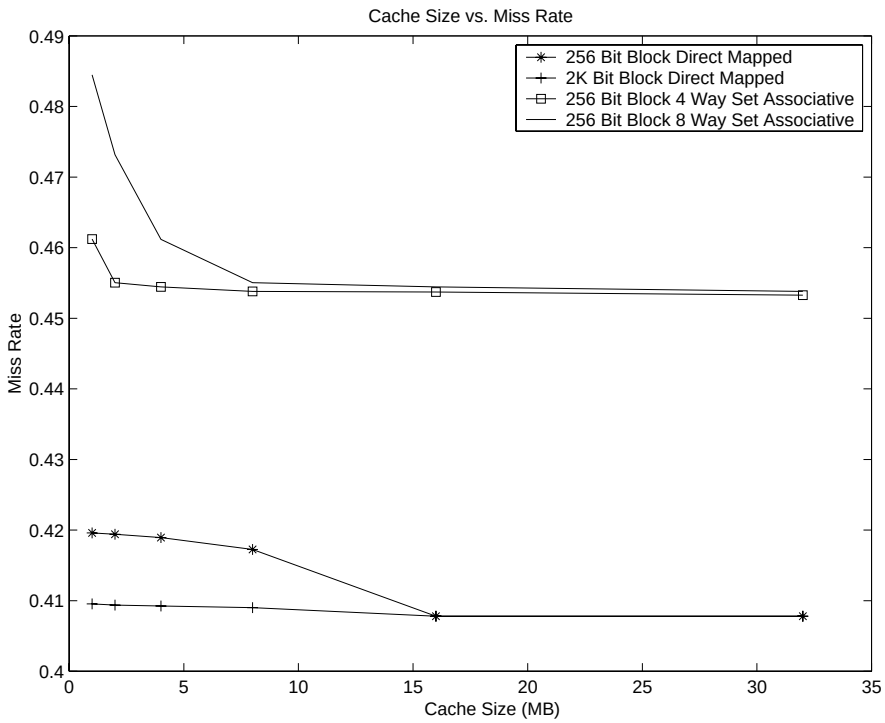


Figure 5.3. DIS FFT Cache Size vs. Miss Rate

5.4.3 DIS Method of Moments

The Method of Moments benchmark also showed very little benefit from increasing the PIM size beyond 1 MB. The 256 bit direct mapped cache proved equally effective to the 2 K-bit cache in terms of miss rates (though it is difficult to see that the two lines are merged in Figure 5.4. However, the 2 K-bit cache demonstrated slightly better performance in terms of run lengths, particularly on the smaller configurations. Figure 5.5 shows the CIPD for the 1 MB cache configuration. The 2 K-bit direct mapped cache has nearly double the probability of the same run length as the 256 bit cache.

As the PIM size grows beyond 1 MB the performance actually degrades. This is best illustrated by the difference in scale between the 1 MB graph (Figure 5.5) and the 32 MB graphs (Figure 5.6). Though the shapes are similar, the 32 MB

configuration shows roughly $\frac{1}{4}$ the probability that a run length will exceed 1,000 instructions as compared to the 1 MB data. This is because the Method of Moments benchmark uses several very large matrices which are competing for the same location within the cache. One would think that the set associative cache would begin to overtake the performance of the 2 K-bit cache. Figure 5.6 does indeed show that the set associative performance improves, however not enough to overcome the spatial locality provided by the 2 K-bit configuration.

These results may seem counter intuitive at first, however, they are the perfect example of the detail which can be lost when examining only miss rate, as well as the behavior of applications which use very large matrices.

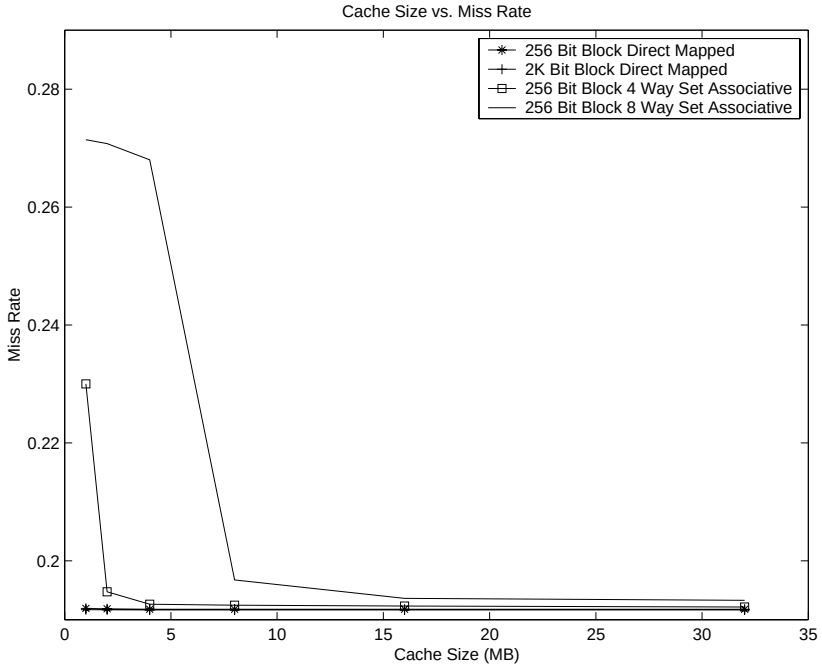


Figure 5.4. DIS Method of Moments Cache Size vs. Miss Rate

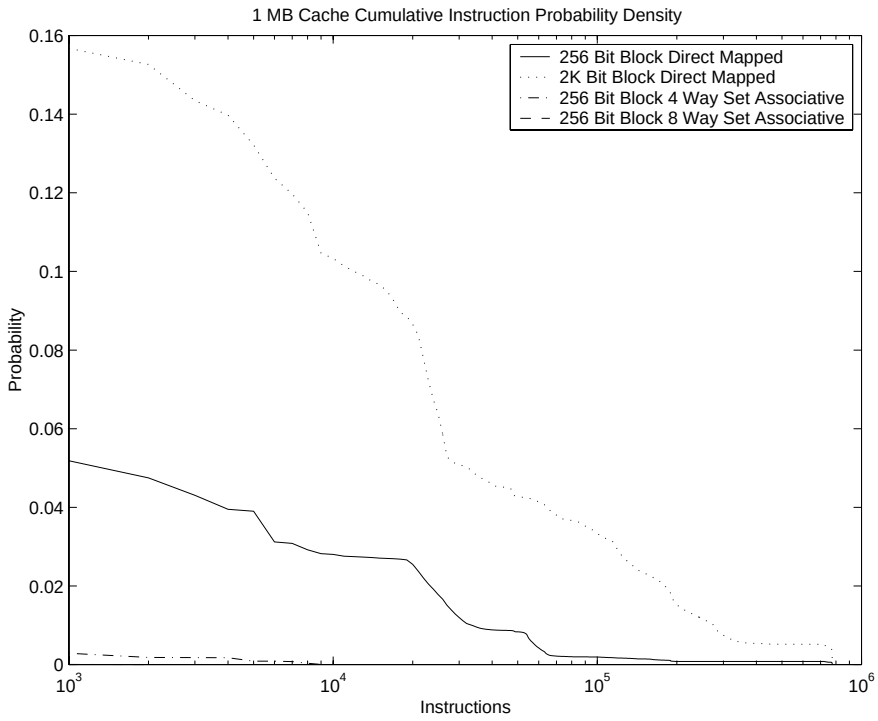


Figure 5.5. DIS Method of Moments 1 MB Data Cache CIPD (Ψ)

5.4.4 DIS Image Understanding

The Image Understanding benchmark, unsurprisingly, showed that the 2 K-bit direct mapped cache provided the best numbers. Again, Figure 5.7 shows that configurations above 1 MB yield virtually no improvement.

5.4.5 DIS Ray Tracing

The DIS Ray Tracing benchmark is the only one where for a small configuration the 2 K-bit cache is outperformed by any other configuration. In this case, the 256 bit direct mapped cache yielded lower miss rates until a PIM size of 4 MB was reached. (See Figure 5.8.)

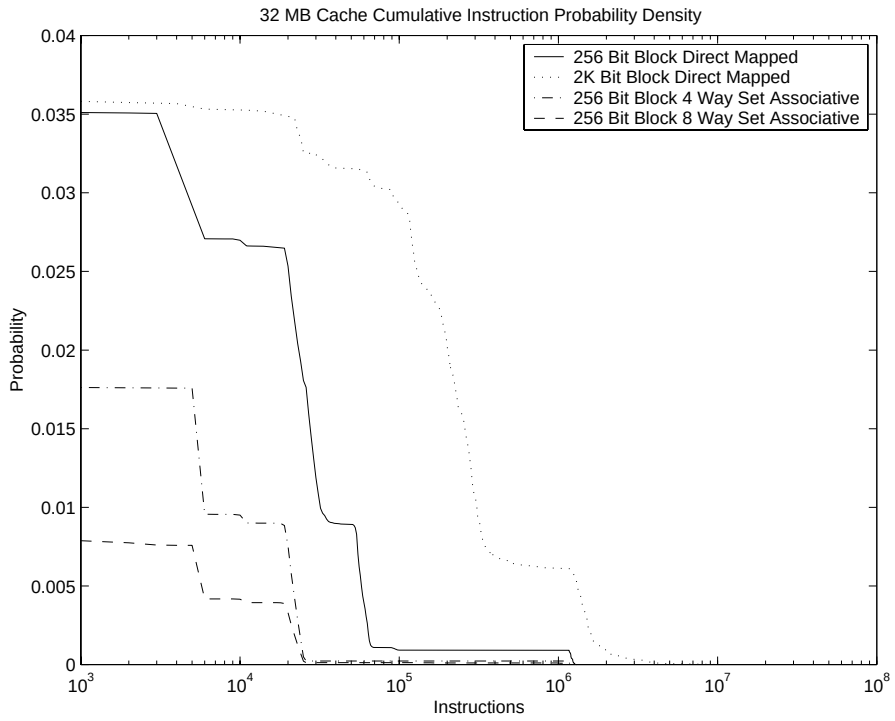


Figure 5.6. DIS Method of Moments 1 MB Data Cache CIPD (Ψ)

5.4.6 Molecular Dynamics Simulation

The Molecular Dynamics simulation confirmed the results of each of the other experiments. the 2 K-bit direct mapped cache was able to provide enough spatial locality to yield significantly better performance than any of the other configurations. Increasing the PIM size beyond the minimum 1 MB, however, did not yield much improvement for the direct mapped configurations. (See Figure 5.9.)

5.5 Page Results

The page experiments examined both data accesses as well as code and stack accesses. The code and stack numbers (all run with 4 KB page sizes) indicated that a very small number of 4 KB pages were needed for each. Access to heap data, not surprisingly, proved to be the dominant factor in achieving significant run lengths. All of the benchmarks studied benefited from larger page sizes, which indicates

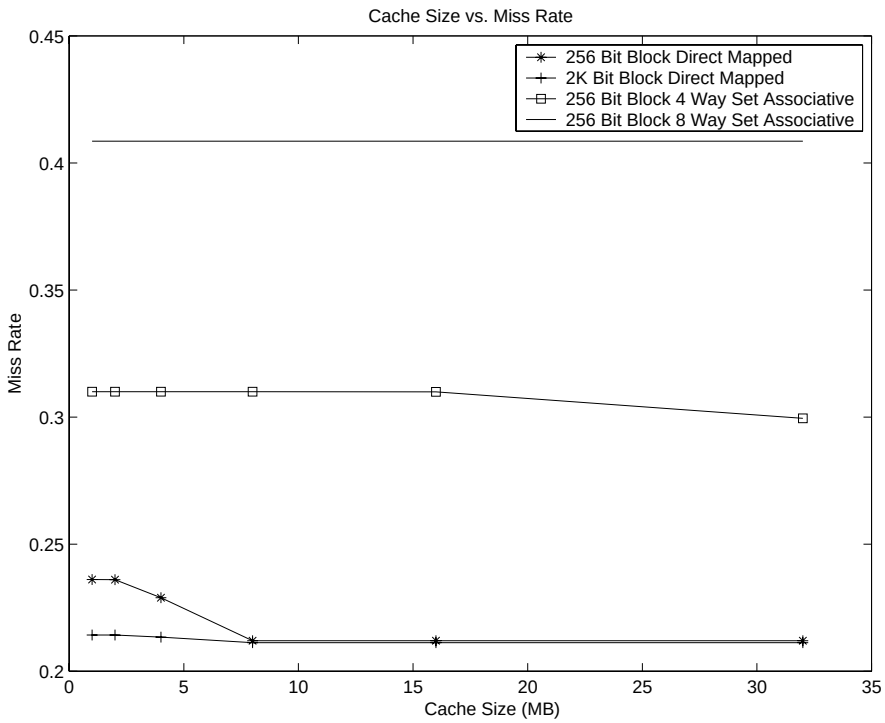


Figure 5.7. DIS Image Understanding Cache Size vs. Miss Rate

that a relatively small number of "windows" into the address space were needed. Similarly, fragmentation problems, which could occur if a large page was accessed infrequently because a section of it was empty, or when there are too few windows into the address space, tended to be insignificant. When accessing larger pages, the benchmarks were uniformly able to take advantage of spatial locality. Furthermore, most of the benchmarks only required a small PIM to achieve significant performance (2, 4, and 8 MB configurations generally fared best). The low reuse inherent in these benchmarks is somewhat advantageous in this regard – data which was quickly streamed through sat unused on larger nodes. This means that the most effectively sized working set is relatively small compared to the data which the program uses throughout its execution.

Because of the large number of configurations derived from the various runs, this section will concentrate only on significant results.

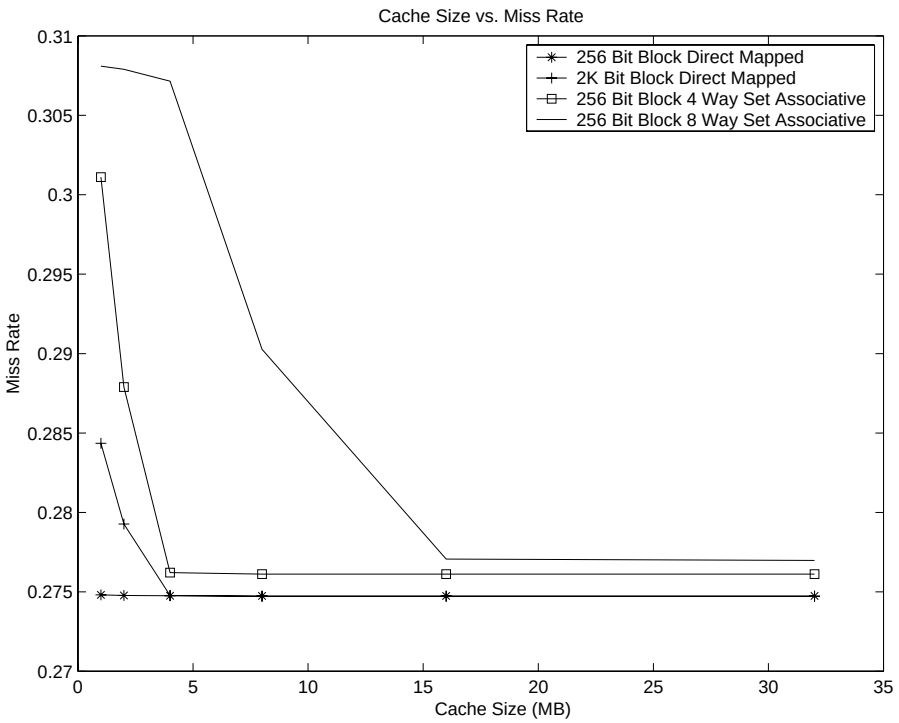


Figure 5.8. DIS SAR Ray Tracing Cache Size vs. Miss Rate

5.5.1 Code and Stack Pages

Generally, the code and stack required a very small area to satisfy the demands of the program. Figure 5.10 is typical of programs with larger code and stack requirements. It shows the DIS Data Management benchmark's results. Many of the benchmarks required only one code or stack page to achieve a miss rate of virtually zero. In this case, one page produced a miss rate of significantly less than 10% on code and 1% on the stack.

Clearly the 4 KB granularity used in this study is relatively large compared to the program's requirements. Chapter 7 will show more fine grain numbers for both the code and stack.

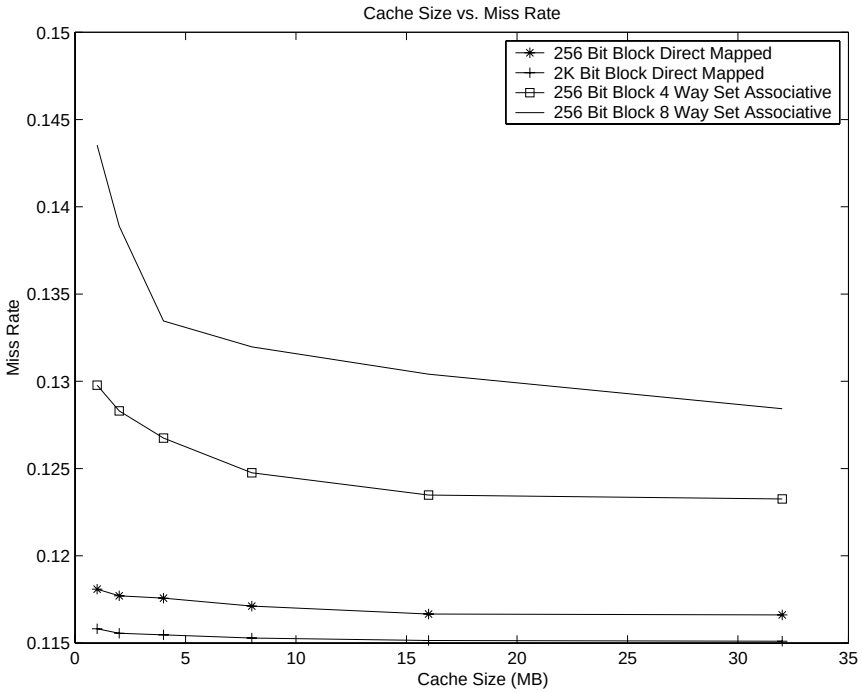


Figure 5.9. Molecular Dynamics Simulation Cache Size vs. Miss Rate

5.5.2 True Data Pages

All of the data miss rate numbers looked very similar to Figure 5.11. For this reason, only CIPD numbers (as in Figure 5.12) will be presented. Unabridged miss rates can be found in Appendix C.

Furthermore, since the run lengths uniformly benefited from larger pages, only the 256 KB page runs will be presented here (since they produced the best results, and have the smallest number of lines to interpret). Additionally, the use of only 256 KB pages allows the analysis to focus on PIM size rather than page size. Again, the unabridged numbers can be found in Appendix B.

DIS Data Management

The results shown for the Data Management benchmark in Figure 5.12 are the most straight forward of any of the benchmarks. The figure clearly shows that between .5

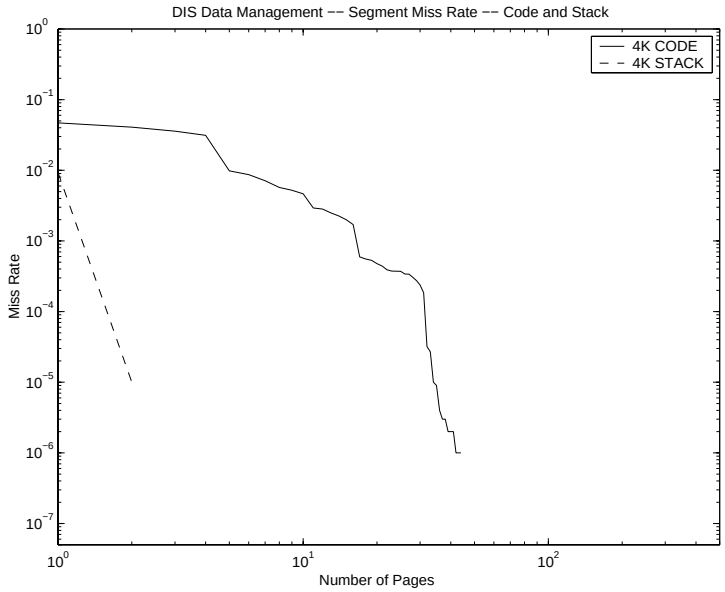


Figure 5.10. DIS Data Management Code and Stack Miss Rate (4 KB Pages)

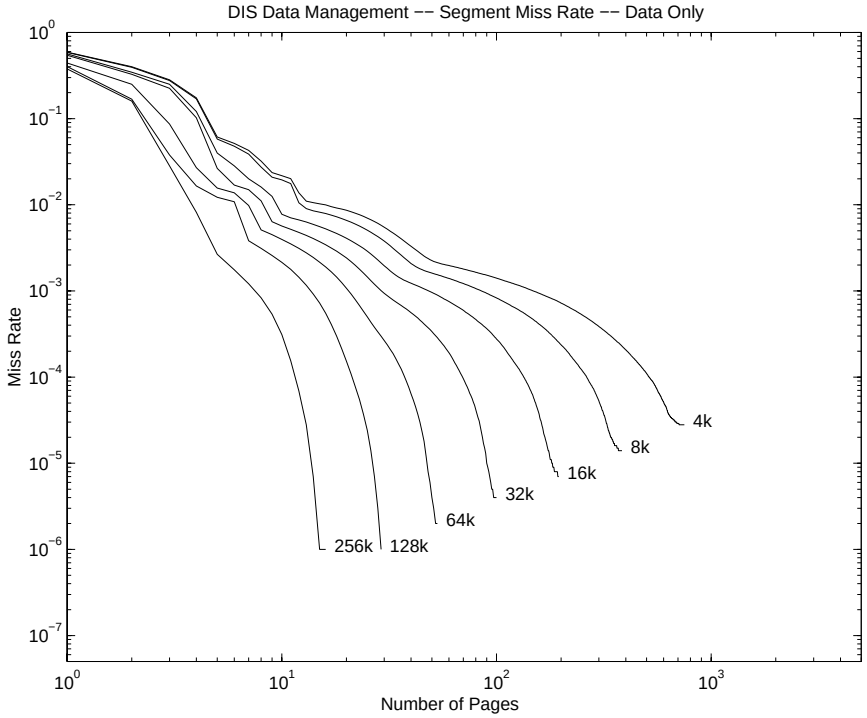


Figure 5.11. DIS Data Management Overall Miss Rate

and 4 MB PIM sizes, the run lengths improve significantly. Furthermore, it shows that for PIM sizes of greater than 4 MB there is no change in the run lengths

generated (hence the lack of lines above 4 MB). Although the benchmark requires significantly more than 4 MB of total data, the vast majority of time is spent in the query, which only has about 4 MB worth of index values. The remaining operation, which is to return the values gathered, is a streaming operation and does not benefit from increased PIM size.

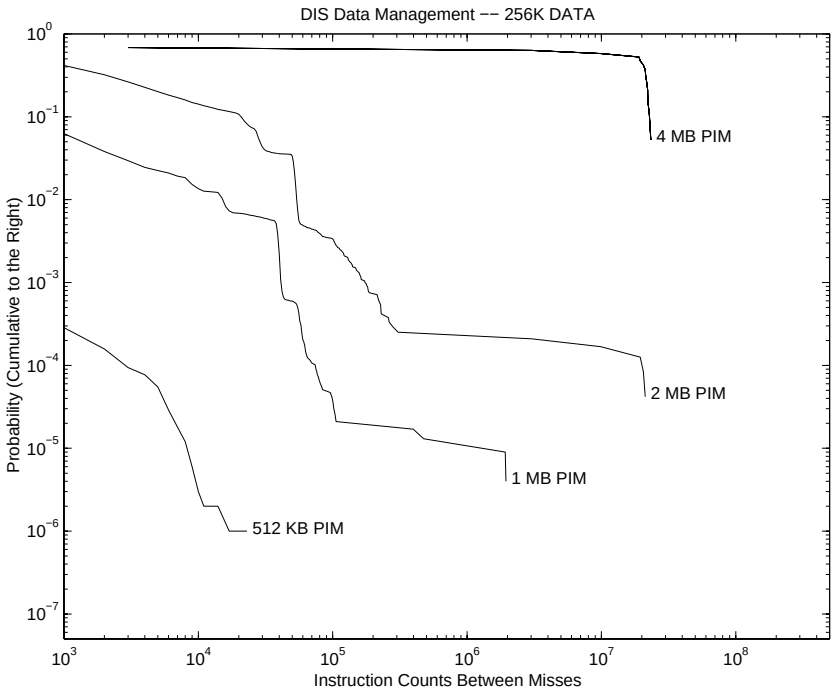


Figure 5.12. DIS Data Management 256 KB Page CIPD (Ψ)

DIS FFT

The FFT showed two distinct areas of interest: the first at 1 MB and the second at 8 MB. Figure 5.12 shows very little difference between the 1, 2, and 4 MB configurations. Similarly, the 8 and 16 MB configurations were also clustered together. Beyond 16 MB no advantage was gained. The gain for an 8 MB PIM is tremendous. There is nearly a 4 order of magnitude increase in the probabilities for all run lengths over about 10^3. In fact, configurations less than 8 MB generally performed

poorly.

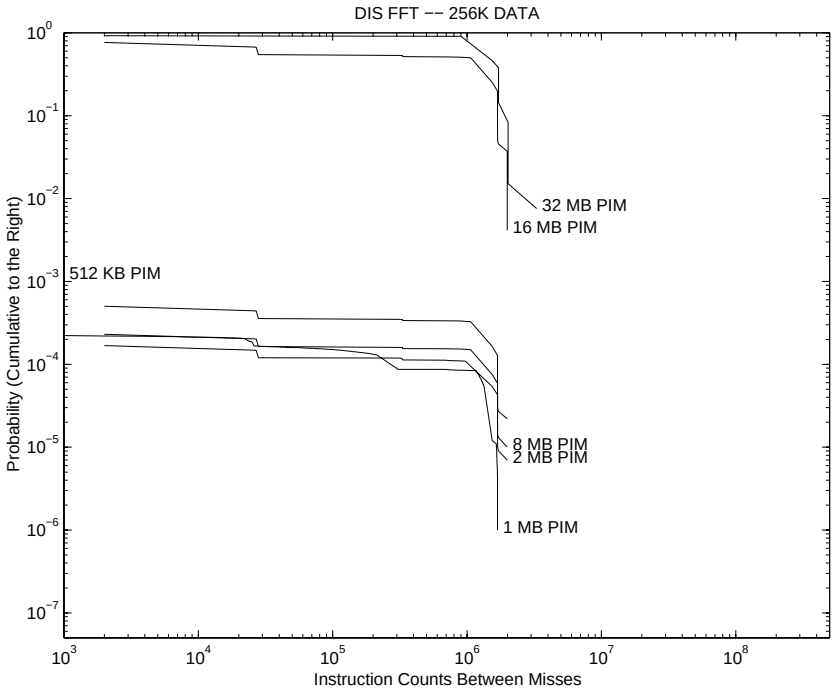


Figure 5.13. DIS FFT 256 KB Page CIPD (Ψ)

DIS Method of Moments

A 2 MB PIM produced fairly high probabilities that run lengths over 10^3 would be generated. Clearly this represents the critical mass for a PIM memory macro size for the Method of Moments benchmark. Figure 5.14 further shows that a 4 MB size produced gains in the overall lengths of many run, but did not significantly increase the maximum run length. Configurations above 4 MB did not yield significant improvement.

DIS Image Understanding

The Image Understanding benchmark required a 4 MB PIM to show significant performance. Although an 8 MB PIM helped to improve the maximum run length

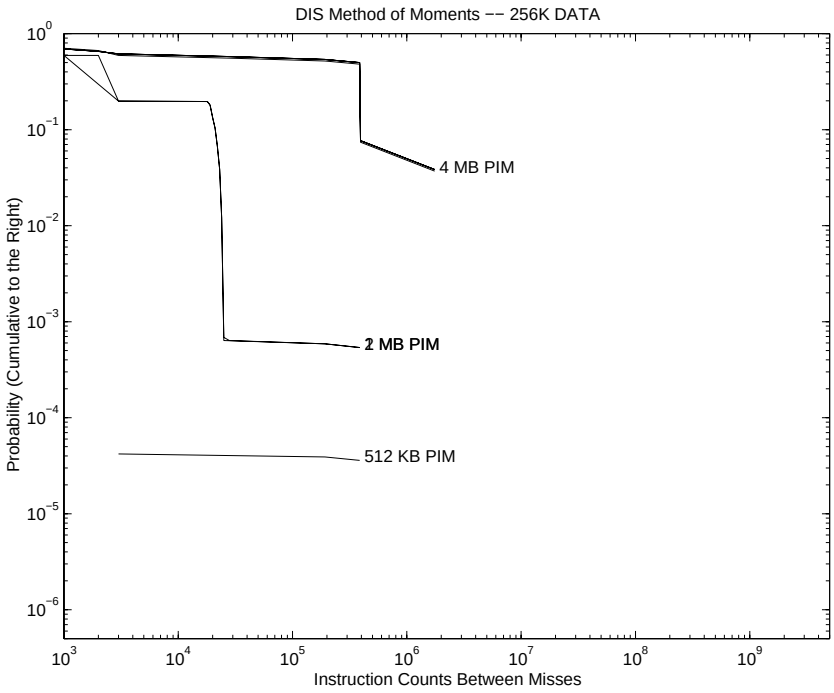


Figure 5.14. DIS Method of Moments 256 KB Page CIPD (Ψ)

by about an order of magnitude, no significant gains were made after 8 MB in size. This result is not surprising considering the relatively large sizes of the images that must be compared (even in the Fourier domain).

DIS SAR Ray Tracing

Though this graph is one of the most difficult to read, PIMs with size less than 2 MB produced a relatively insignificant number of runs greater than 10^3. The 2, 4, and 8 MB configurations were similar, except that larger sized PIMs generally increased the maximum run length significantly (from about an order of magnitude for each doubling of the PIM size). Beyond 8 MB, changes in PIM size proved less relevant. See Figure 5.16 for details.

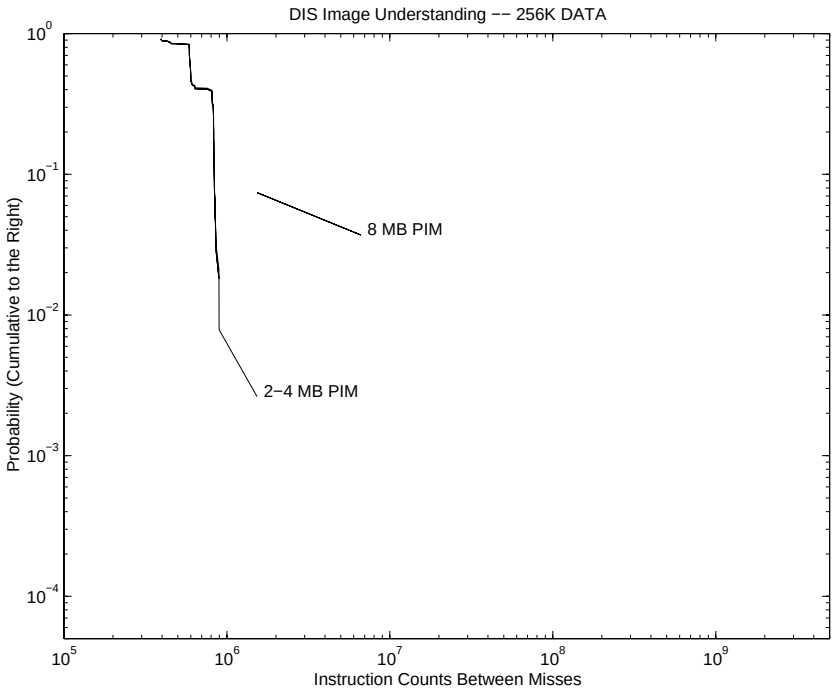


Figure 5.15. DIS Image Understanding 256 KB Page CIPD (Ψ)

Molecular Dynamics Simulation

Figure 5.17 shows that a 2 MB PIM clearly achieves significant performance for the Molecular Dynamics Simulation. In fact, of all the benchmarks examined, this simulation showed some of the longest overall run lengths. Significant gains were made in terms of the maximum run length by increasing the PIM size to 4 MB, however, beyond that only small improvements appeared.

5.6 Summary of Results

Tables 5.1 and 5.2 show the mean and median values (respectively) of the CIPD for 256 KB pages.

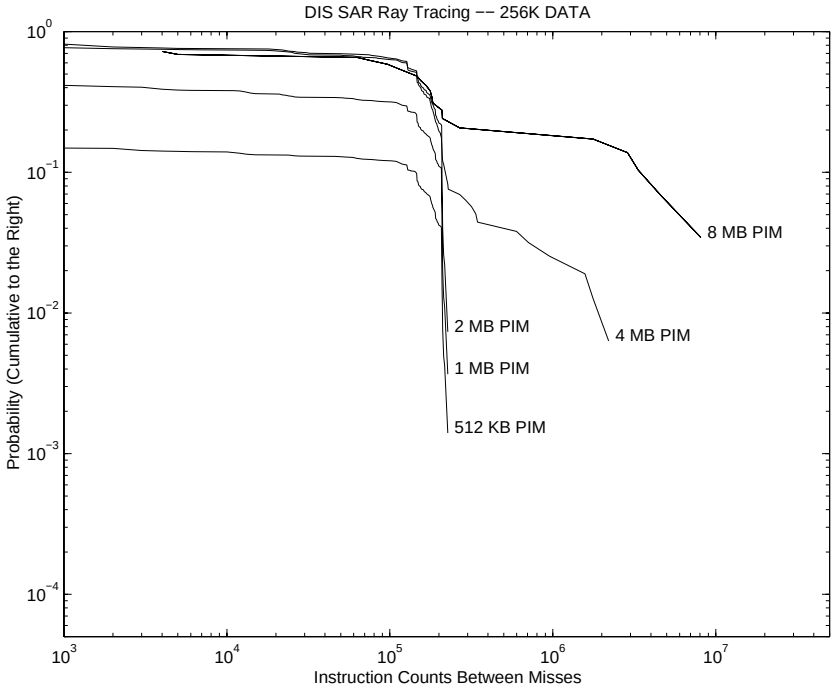


Figure 5.16. DIS SAR Ray Tracing 256 KB Page CIPD (Ψ)

5.7 Conclusions

Programmers do indeed allocate even pointer-based data in a relatively uniform fashion [33]. This allows for tremendous gains in performance through the use of spatial locality. For this reason, larger page sizes, as well as larger cache lines yielded the greatest performance increase. Furthermore, the page configurations, being better able to take advantage of this property, outperformed the cache configurations (generally by 1 to 2 orders of magnitude).

The direct mapped caches, particularly with large cache lines, always outperformed the set associative caches, however, they were generally unable to take advantage of memory macro sizes greater than 1 MB. PIMs, as well as modern microprocessors with large on-chip L2 caches, will achieve better performance by using on-chip memory as a page space.

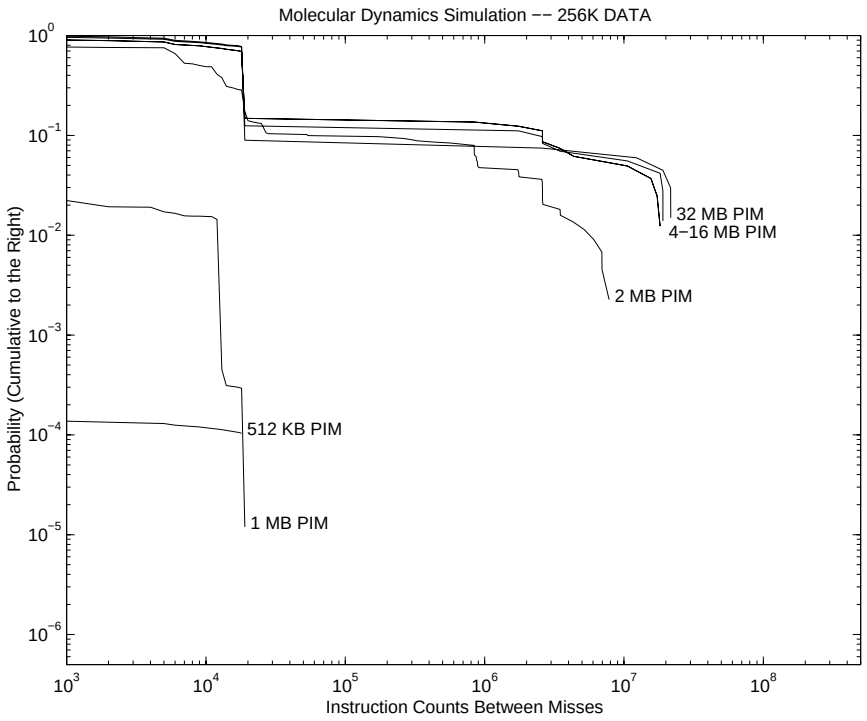


Figure 5.17. Molecular Dynamics Simulation 256 KB Page CIPD (Ψ)

Increasing the page size did not generally lead to fragmentation problems. In fact, a 2 MB PIM with 256 KB pages, which only has 8 "windows" into the overall address space, achieved critical mass for every benchmark except arguably the FFT which required an 8 MB configuration to achieve significant run lengths (greater than 10^3). In general, most benchmarks did not improve significantly beyond the 8 MB configuration due to the relatively large amount of streaming.

Table 5.1. Working Set CIPD Mean Values (256 KB pages)

BENCHARMK	**2 MB**	**4 MB**	**8 MB**	**16 MB**	**32 MB**
DIS DM	10,000	13,158,000	13,158,000	13,158,000	13,158,000
DIS FFT	200	300	600	1,032,700	1,893,700
DIS MoM	5700	9,258,900	2,615,200	9,615,200	9,615,200
DIS IU	632,900	655,700	9,259,100	9,259,100	9,259,100
DIS RAY	117,400	202,600	1,105,900	1,105,900	1,105,900
MD	225,700	1,233,900	1,233,900	1,388,200	1,491,800

Table 5.2. Working Set CIPD Median Values (256 KB pages)

BENCHARMK	**2 MB**	**4 MB**	**8 MB**	**16 MB**	**32 MB**
DIS DM	2,000	20,549,000	20,549,000	20,549,000	20,549,000
DIS FFT	2,000	2,000	2,000	1,540,000	1,715,000
DIS MoM	18,000	393,000	393,000	393,000	393,000
DIS IU	601,000	601,000	1,531,000	1,531,000	1,531,000
DIS RAY	148,000	149,000	155,000	155,000	155,000
MD	11,000	843,000	843,000	1,749,000	2,604,000

CHAPTER 6

COMMUNICATION AND DATA DISTRIBUTION

We few, we happy few, we band of brothers...

– King Henry, *Henry V*, William Shakespeare

Thus far, this work has examined PIMs as uniprocessors. This chapter will begin to examine the properties of a PIM multiprocessor (see Figure 6.1). The execution of programs on multiprocessors is dominated by the overhead of communication. This overhead is particularly important on PIM systems because individual PIM nodes are less powerful than their counterparts on modern parallel systems. This produces a twofold problem: first, more nodes are required to create a memory capacity of equivalent size (making the interconnection network bigger); and second, there is less local storage available per processor for data (meaning that generally more communication requests will be generated). Therefore, this chapter will quantify the cost of communication for the given benchmarks for different interconnection network topologies.

The communication overhead is measured in terms of the distance between communicating nodes (in "hops"). The longer the distance, the more overhead a given communication will incur. Three common interconnection networks are used: a ring, a 2-d mesh, and a binary hypercube. Since innumerable configurations for a communication network possible, a more generic measurement of communication overhead must be made. This measurement, known as the *communication radius*

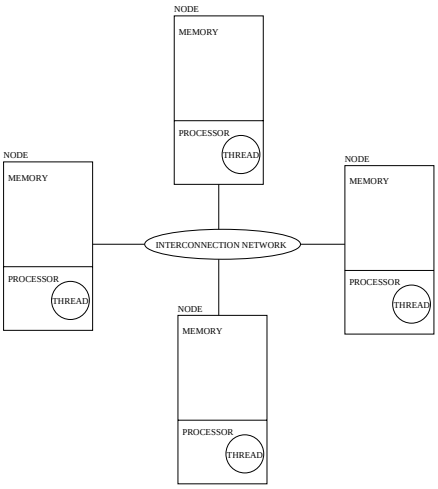


Figure 6.1. PIM as a Multiprocessor

indicates with how many unique nodes a given node communicates over the course of a program. If a network topology could be constructed for each benchmark, which would place all nodes that communicate with each other one hop apart, and which would eliminate any unused connections, the communication radius for each node indicates the number of connections that begin at that node.

It will be shown that over the course of each of these benchmarks, a given node communicates with a small number of other nodes. Given the right network topology, data on different nodes can be placed physically close to the next piece of data to be accessed, which serves to lower the communication cost. Since the communication radius is small, this goal can be achieved at relatively low cost. Furthermore, experimentation shows that the mobile thread execution model (described in Section 6.1 allows a mobile thread to move from node to node, with relatively long run lengths on each before another movement is required. This serves to amortize the cost of communication.

6.1 Parallel Execution Models

Chapter 5 indicates that a given PIM node can sustain significant computation by accessing only the contents of its working set. This is primarily how all parallel machines function. The working set in a CC-NUMA or COMA machine is the data contained on a particular node. Message passing systems expose the function of updating the working set to the programmer. For the Tera it is the contents of a thread's register file.

In each case, execution proceeds over the working set until the working set must be updated or altered. When that happens, execution stops (at least for the thread which is attempting to access data outside working set) until the set can be updated. The expense of updating the working set is defined by each architecture. On more traditional parallel machines, that cost is driven by the speed of the interconnection network and coherency hardware. The Tera ameliorates this cost by allowing another thread to execute in the place of the blocked thread. However, on a very large PIM system the sheer scale may make it very difficult for the program to provide enough threads to keep this cost low. Furthermore, there is ample research available to describe the implementation and price of these methods.

The proposed method of updating the working set for the purposes of this chapter is the *mobile thread*, or *rolling snowball*. Rather than bringing the needed data to a thread which requests it, the thread's execution is brought to the data. There are several advantages to this approach in the context of PIM. Most importantly, no coherency protocol is required because the execution of code is tightly coupled with the data which that code requires (ie, the number of copies of any given piece of data floating around the system is minimized by forcing most code which modifies that data to execute on the node which "owns" the data).

Furthermore, PIMs are designed to take advantage of local memory accesses.

Without a high degree of locality PIM as an architecture simply makes no sense. Therefore, the ideal situation is for a thread to execute for a very long period of time on one node then move on.

In the case of each benchmark, only one thread is examined – the "main loop" of the program. This represents a worst-case demand for resources (lighter weight threads should require less computation time, less data, and fewer communications). It is expected that programs written specifically for parallel multi-threaded computation will produce better results than those presented here.

6.2 Data Placement

One of the most difficult problems in parallel computing is deciding upon an effective data placement policy. Specifically, it consists of choosing an efficient physical location for each data structure which the program accesses. Fortunately, programmers tend to allocate data structures with a very high degree of internal spatial locality [33].

Given the mobile thread model, and the results from Chapter 5 (that large page sizes are very useful), the data placement chosen for these experiments is to divide the address space into large "chunks" of memory that are the size of an individual PIM. (The familiar 2, 4, 8, 16, and 32 MB PIMs are all examined.) No special rearrangement of data is made for this simulation – it appears in whatever sequence the programmer allocated it.

This represents an extremely simple PIM compiler and run time system. Consecutive requests to allocate memory are serviced contiguously.

6.3 Communication Costs

When a mobile thread misses data on the local node, some movement across an interconnection network is made (either the movement of data or code). The number of *hops* required for a given communication to occur is the number of nodes through which the message must pass to move from the source to the destination, using a given routing algorithm. Two communication cost measurements are made in this experiment each time communication is attempted: first, the actual cost of moving (in terms of hops) from one node to the next; and second the overhead cost of determining where to route the message. Since it is assumed that a given memory address can be arbitrarily located throughout the system, when a remote name is encountered it must be translated to a physical location.

Figure 6.2 shows four possible name resolution mechanisms which can be used to determine a route for the thread which must move (or data which must be found): first, the node which generates the request in need of resolution knows, *a priori*, that resolution; second, another node (which is known to the node in distress) can resolve the name; third, the interconnection network (deterministically) knows how to resolve the name; or forth, the name can be broadcast to all nodes and the node which possesses that name will respond. The first method, which consists primarily of maintaining a page table on each node, is discussed in Chapter 4. The second method, which consists of maintaining a *dictionary node* to store name resolution information, will be examined in this chapter. The third (deterministic) method is assumed to incur zero overhead. The fourth is assumed to be equivalent to the cost of a broadcast, which is well known for most interconnection networks. That means that only the second method, specifically requesting the resolution of a name from another (dictionary) node will be examined.

It should be noted that these four methods can be combined in virtually any

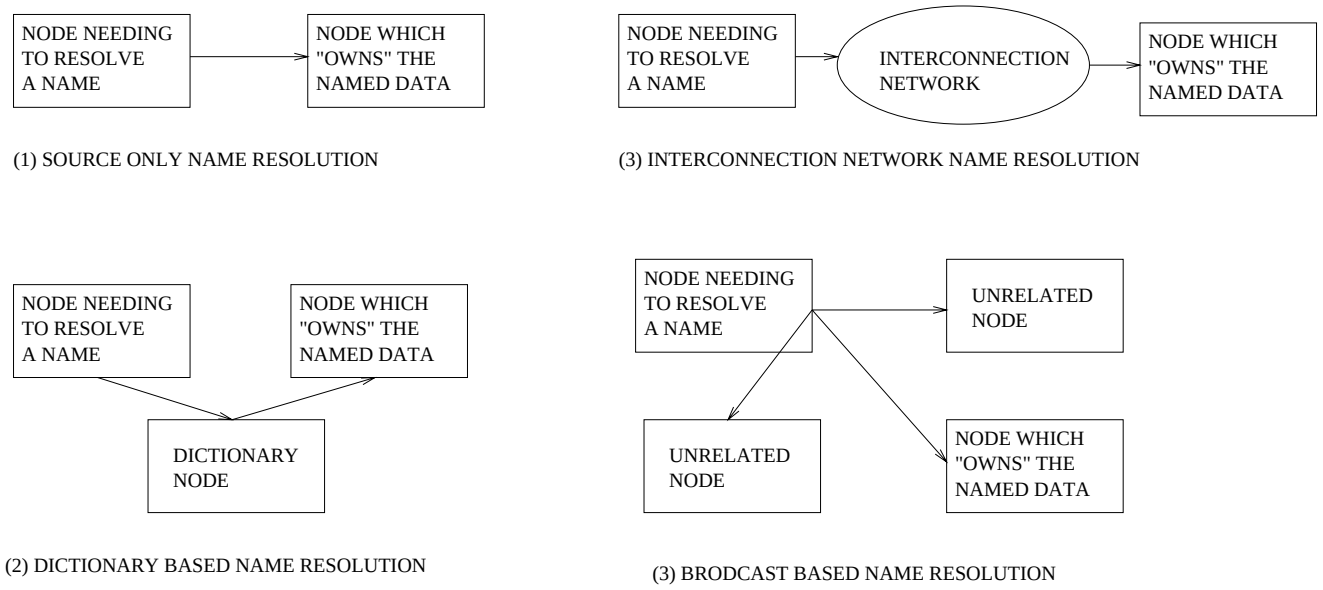


Figure 6.2. Four Possible Translation Mechanisms

combination of ways. For example, a node may maintain a frequently used set of name resolutions (similarly to the maintenance of a TLB) and use a *dictionary node* or a broadcast to resolve unknown names.

Translation at the node which encounters the remote reference, hence forth termed *source translation*, can occur by two mechanisms. First, each node may contain translation information about every address in the system. This would, however, require a very complex update process if data were to be moved. Furthermore, if the page size is relatively small, a large table must be maintained on each node (see Chapter 4 for information about the size of such a table).

The other possible *source translation* mechanism is through the use of a small cache of frequently translated names (similar in function to a TLB). This does not guarantee that a name presented to a node can be translated locally, therefore some sort of fall back mechanism must be in place. (See Section 6.11 for further information.)

Finally, translation by means of some intermediate node which contains a com-

plete dictionary is performed by asking that dictionary node where a given address resides. This involves an extra series of hops to get to the node holding the dictionary.

The cost of name resolution represents the key measurement in understanding the overhead generated by a particular network configuration. This cost is primarily dominated by any extra communication which must occur, or any "overhead" space which a node must use to maintain tables. Since a memory address on PIM systems could be off chip, determining the exact node upon which that address resides is a difficult problem. For the purposes of this experiment, it is assumed that the directory is contained on 4 nodes evenly distributed across the interconnection network. The measured translation cost represents the upper bound on overhead, as it assumes that a given node must ask the directory each time an off-chip communication is necessary.

6.3.1 Interconnection Networks

The three interconnection networks simulated are: a 2-d mesh, a ring, and a binary hypercube [31]. Each network is assumed to consist of up to 2048 nodes. Thus, the mesh used is 32x32, the ring is 2048 around, and the hypercube is 10-dimensional.

All communications are presumed to have occurred "ideally" given the routing algorithm. On the ring, this means that the choice of going left or right is assumed to have always been made correctly, and on the hypercube, minimal cost routing is used [20]. The mesh uses the simple X-Y routing scheme (that is, movement is first made in the X direction and then in the Y), which produces "L"-shaped movement each time) [20]

While knowing the communication patterns on specific interconnection networks is useful, only a small number of networks could be analyzed due to limitations on

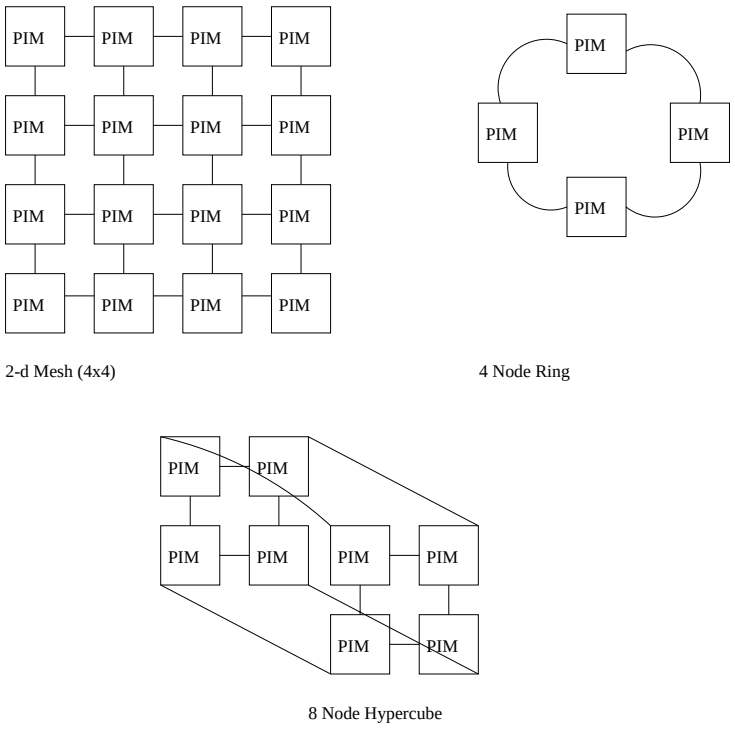


Figure 6.3. Example Interconnection Networks

simulation time and the number of results which can be presented. In a more general sense, understanding how many unique nodes a given node will communicate with is more revealing. The *communication radius*, which is defined as the number of nodes with which a given node communicates, measures precisely that. Thus, the *communication radius probability density*, which is the probability that a given node will communicate with n or fewer nodes, is also presented. This measures the number of unique nodes with which a given node is likely to communicate.

6.4 Experimental Configurations

Aside from the three interconnection networks used to measure the actual routing from one node to another, two policies for thread movement are tested. The first policy dictates that a thread executes until **any** remote access is encountered. The thread then stops, determines which node it must move to next, and moves. The

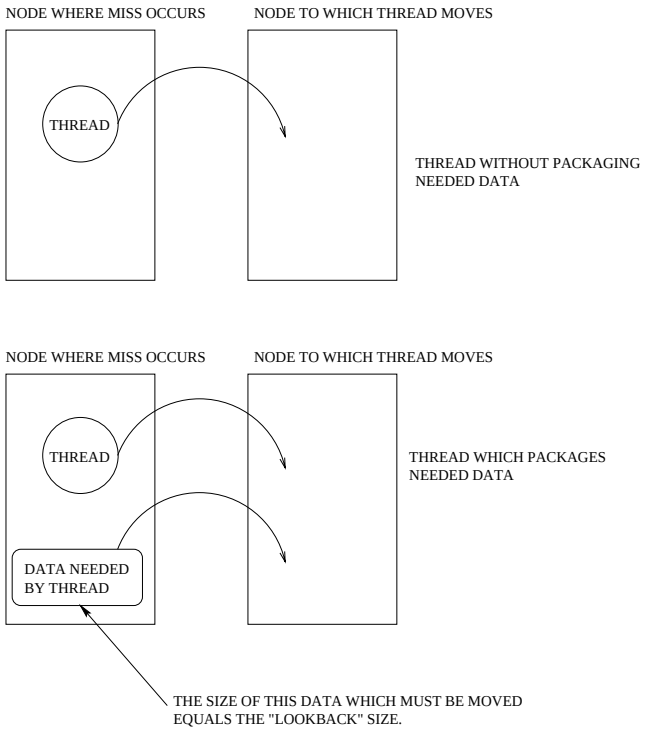


Figure 6.4. The two types of thread movement

process is then repeated. This model assumes no data is carried with the thread. At the next access (even to the prior data), a miss must occur and the thread moves again.

In the second policy, once a thread moves, it is allowed to access data located on the previous node without triggering a miss. Each address on the previous node which the thread touches is tagged by the simulator, and when the thread moves again the number of unique addresses which were touched on the previous node is computed. This represents the amount of data that could move with the thread to cause no references to the previous node. These types of references are referred to as *look-back* references. This information gives some idea of *context size* when the current thread moves from one node to the next. See figure 6.4. (Chapter 7 describes a mechanism by which these look-back references can be captured before the thread moves.)

6.5 Experimentation

The Shade simulator used for experimentation in this chapter tracks several things for each PIM size and thread movement policy. For the purposes of this experiment, only data references are examined. Chapter 7 provides a detailed explanation of how code and stack references should be dealt with.

The simulator tracks every instruction executed in the program. When a memory reference (load or store) to the Data segment is encountered (that is, any memory reference except an instruction fetch or one to the stack), each configuration of the experiment is presented with the address of the reference and the number of instructions which were executed from the previous reference. Each of the configurations uses this data to update its run length and overhead statistics (since PIMs of different sizes will have different segments of the address space contained locally).

The run length information is updated by the number of instructions since the previous reference.

If the current memory reference is remote (taking into account the thread movement policy), the current run length counter is placed into a bucket and reset to zero. The thread must then be moved to the remote node. The overhead for that movement is computed. The overhead of asking a dictionary is determined by computing the round trip communication cost of accessing the dictionary which is physically closest on the interconnection network. It is assumed that there are 4 evenly spaced dictionaries through out the network. The cost for a local translation, in the form of a TLB-like mechanism, is computed to be the miss rate of the TLB for the current node (each TLB has between 1 and 32 entries). The thread then moves to the new node. When movement occurs, the routes for that movement are computed for every interconnection network examined, which accounts for the communication cost in terms of hops.

If the memory reference was local because the thread movement policy allowed looking-back to the previous node, the address of the look-back reference was examined to determine if this was the first request for that data from the current run. All first requests incremented the counter of unique look-back references. Whenever a thread moves, this counter is placed into a bucket along with the run length information to determine the size of look-back references.

6.6 Run Length Data

As in Chapter 5, the success of each parallel implementation is measured in terms of run lengths between misses. The Cumulative Instruction Probability Density graphs (CIPD, or $\Psi(L)$, described in section 5.2) is used to measure run lengths for each experiment in this chapter as well. In this case the CIPD represents the probability that a given thread when it arrives on a node will execute L instructions or more.

6.6.1 DIS Data Management

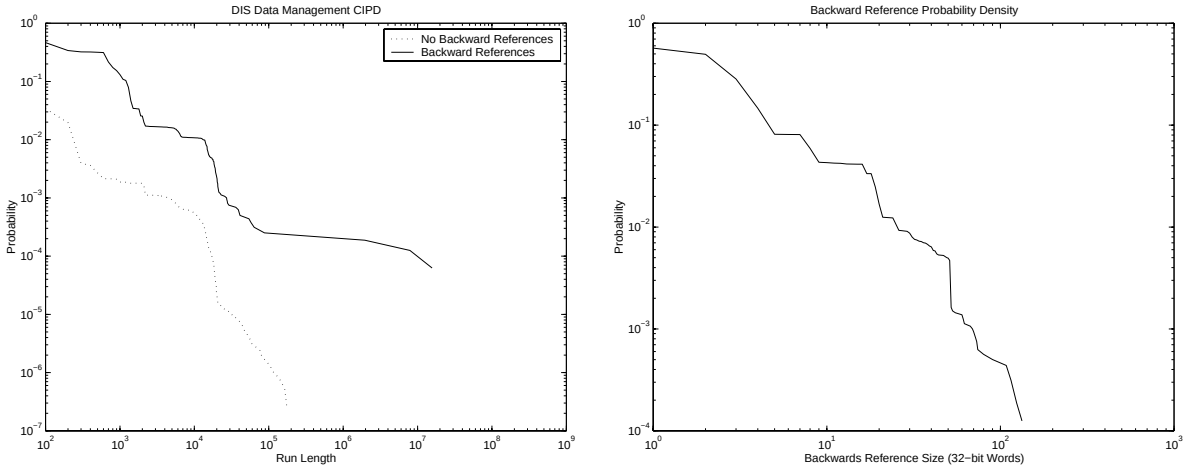


Figure 6.5. DIS Data Management Results (CIPD, $\Psi(L)$)

Generally, this benchmark proved the easiest to quantify in terms of parallel performance. Since the the data structure being traversed is a tree, and the size of

each node is relatively small, the probability of a parent and child being co-located is quite high. Repeated insertions and deletions into a preexisting tree would alter this fact significantly, however, true parallel code would attempt to keep parents and their children located together.

Figure 6.5 shows both the run length and the backwards reference size (in 32-bit words) for the Data Management benchmark.

As is typical for the benchmarks presented, the backwards reference size was not large (nothing was greater than $O(400)$ bytes), however the ability to reference that small amount of data improved performance tremendously. If it were packaged when the thread moved, the probabilities of hitting longer run lengths improved by nearly an order of magnitude. Additionally, the maximum possible run lengths were extended by over two orders of magnitude from about 10^5 to slightly over 10^7. The cost of packaging less than half a kilobyte of data is insignificant compared to those gains.

6.6.2 DIS FFT

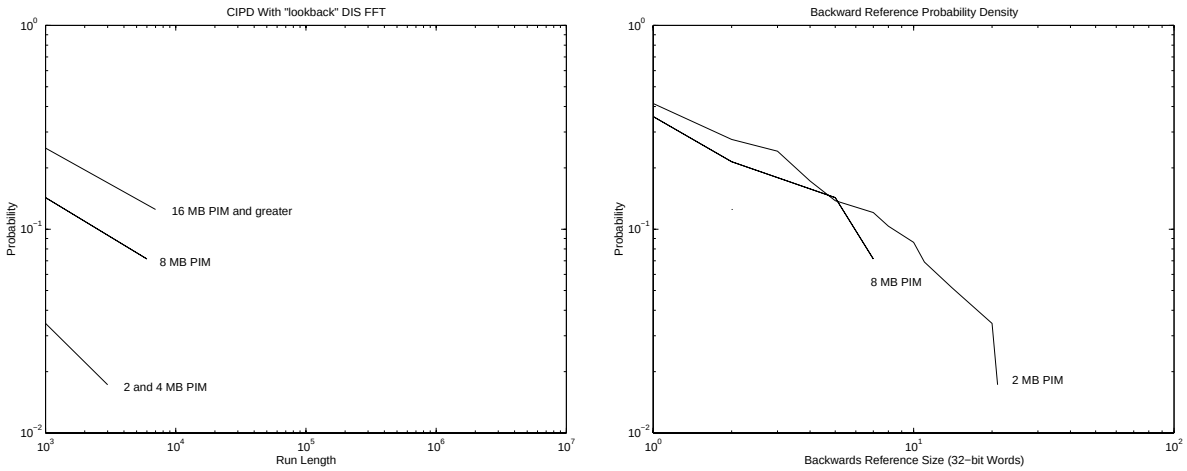


Figure 6.6. DIS FFT Results (CIPD, $\Psi(L)$)

The FFT benchmark was, as in Chapter 5, characterized by short run lengths.

Though the probability of executing more than 1000 instructions was reasonable, especially for the larger configurations, the run lengths never exceeded 10^4. Without the backwards reference ability, in fact, the run lengths were too small to display graphically. However, for a very small amount of data (no more than 40 bytes), the run lengths were improved to the far superior numbers shown in figure 6.6.

There are two very important characteristics of this benchmark which are encouraging for future parallelism. Since the data to be allocated was not interleaved on the PIM in any intelligent way, this represents the absolute lower bound numbers for any implementation. In fact, a compiler today could take the code and make it capable of running on a PIM system without any modification.

6.6.3 DIS Method of Moments

Again, the Method of Moments proved itself very difficult to satisfy and one of the most complex sets of results to analyze. This is not surprising considering that a true parallel implementation is more difficult (particularly in terms of data placement) than the simplifying assumptions that are made in this work. However, the performance curves are not prohibitively bad. The probability of execution for more than 1000 instructions is relatively low. Again, this is because the matrices are very large and are allocated one after the other instead of being interleaved or subdivided.

Without successfully packaging the look-back references there was absolutely no change between configurations. That is, the 2, 4, 8, 16, and 32 MB configurations all produced the exact same CIPD graph. Allowing look-back to the previous node allowed the 4, 8, 16, and 32 MB configurations to improve significantly the probability of executing longer instruction streams between misses, however the maximum run length remained the same. See figure 6.7.

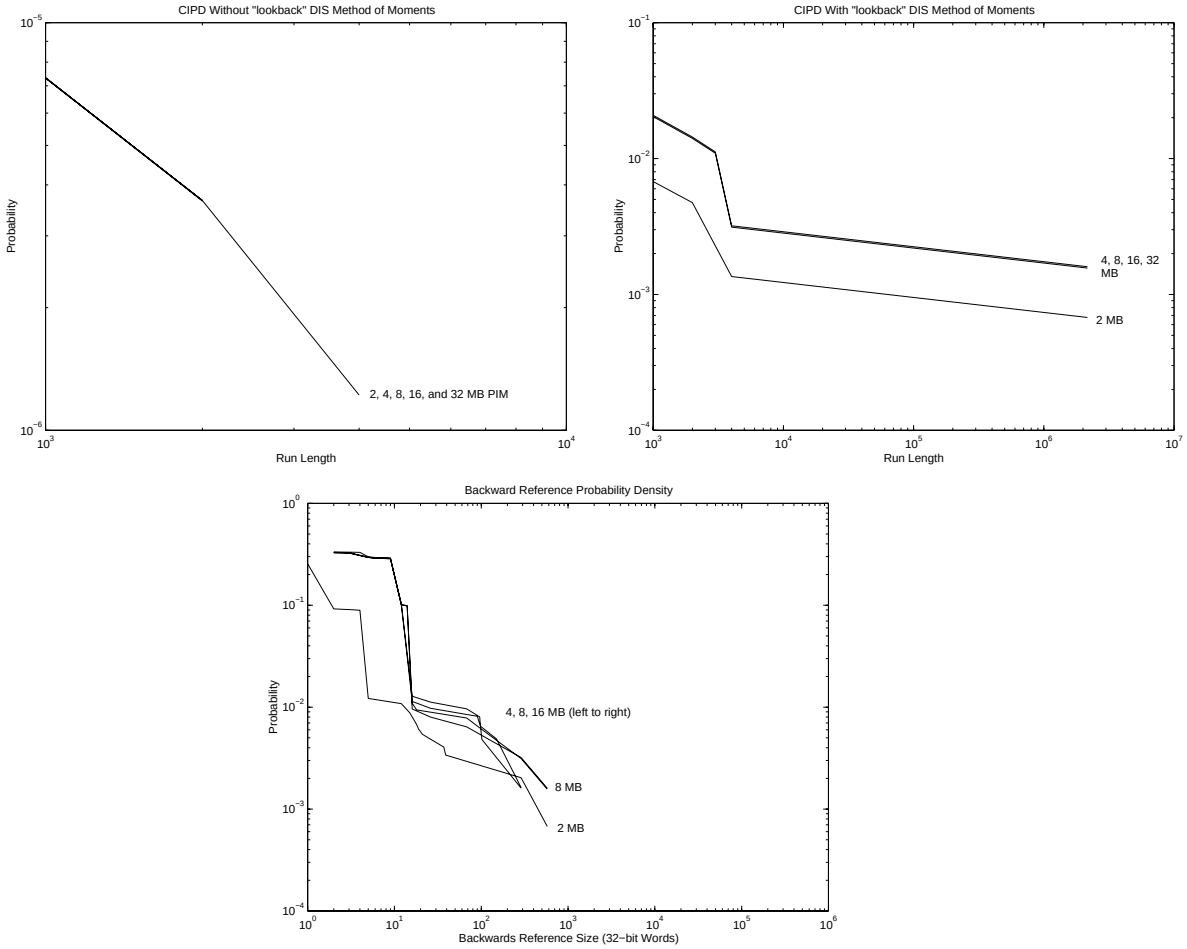


Figure 6.7. DIS Method of Moments Results (CIPD, $\Psi(L)$)

The requests for data from the previous node did change depending on the configuration, though not significantly (see the second graph in figure 6.7). The amount of data which needed to be transferred was moderate – no more than 8KB, however the performance gain was tremendous. Over 3 orders of magnitude were gained on the run length probability densities. This represents a striking improvement at the cost of transferring a small amount of data.

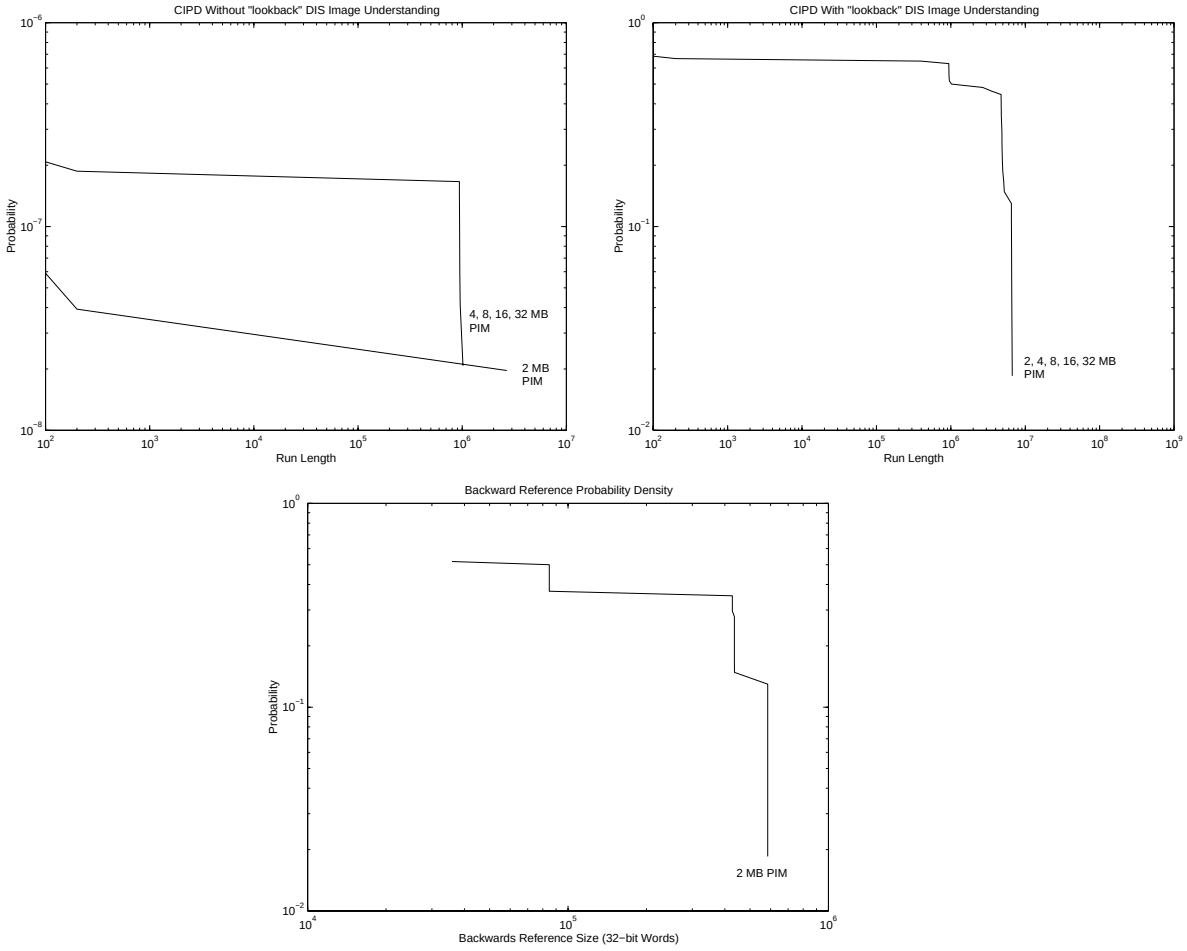


Figure 6.8. DIS Image Understanding Results (CIPD, $\Psi(L)$)

6.6.4 DIS Image Understanding

As with the results in chapter 5, the Image Understanding benchmark again proved the most demanding. The extremely complex nature of both the algorithms and data structures requires that a true parallel and multi-threaded implementation be studied further.

Figure 6.8 shows that without the ability to look-back at previous references, the probability of executing more than 100 instructions without generating a miss was significantly less than 10^{-6}. The ability to look-back proved extremely successful at increasing the probability of a high instruction run, however, the cost was relatively

high. Nearly 512K of data would be required to achieve any result. However, it should be strongly noted that the improvement of the probability of long runs is over 6 orders of magnitude.

6.6.5 DIS Ray Tracing

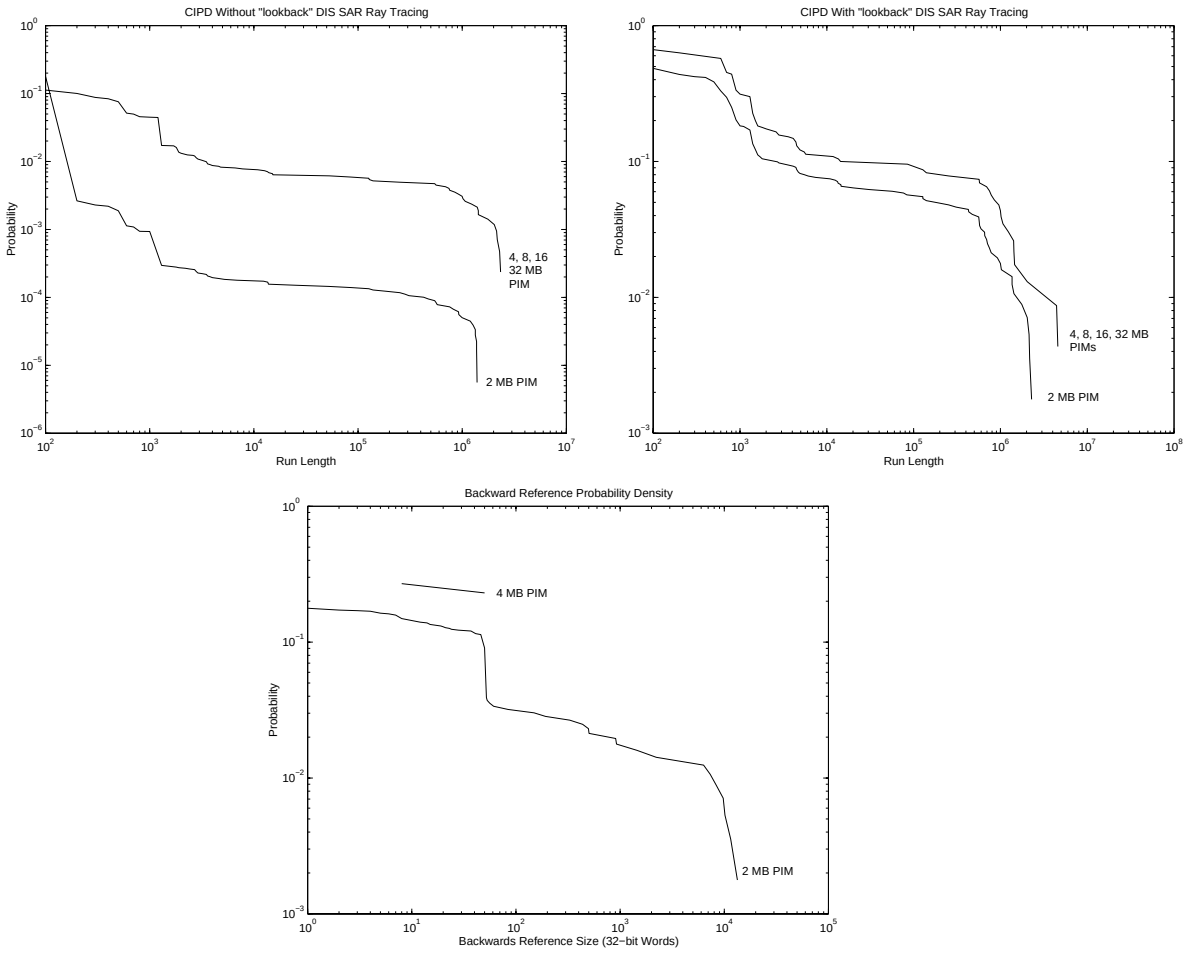


Figure 6.9. DIS Ray Tracing Results (CIPD, $\Psi(L)$)

The DIS Ray Tracing was one of the most successful in terms of run lengths. Even without the look-back optimization, it performed relatively well. There was no difference between the 4, 8, 16, or 32 MB configurations because the working set was effectively captured.

The ability to look-back proved less effective at the 2 MB configuration since it required significantly larger amounts of data to succeed than at the 4 MB configuration (see figure 6.9 for further details). The 4MB runs required less than 1KB to more than double the performance.

6.6.6 DIS Molecular Dynamics

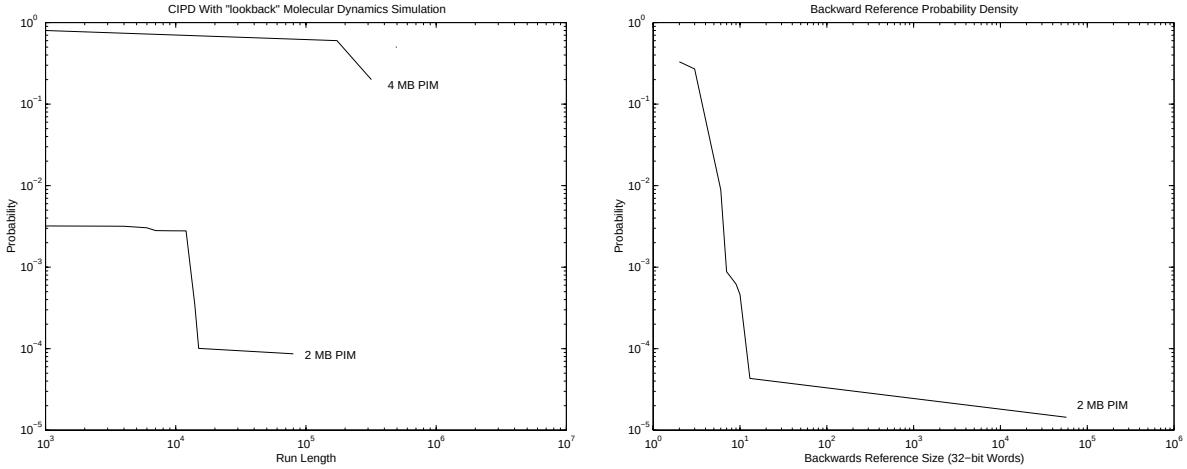


Figure 6.10. Molecular Dynamics Results (CIPD, $\Psi(L)$)

The Molecular Dynamics Simulation exhibited the best performance with the 4 MB configuration. In fact, larger configurations made no difference in performance whatsoever (see figure 6.10). The 2 MB configuration, on the other hand, performed very badly (nearly 2 orders of magnitude worse in terms of probability density). Additionally, while the larger configuration required virtually no look-back, the 2 MB system required nearly 512K for a successful run.

Considering that without the look-back the same 2 MB configuration failed to achieve any instruction runs over 100 in length, the look-back was highly successful, however increasing the macro size proved significantly more effective.

6.7 Communication Radius Data

The *Communication Radius* represents the measure of connectivity between nodes given a "virtual" network with no assigned topology. If, for example, node X exhibits a Communication Radius of 4, that means that there are only 4 nodes in the entire network with which node X ever communicates during the course of program execution. The ideal network for any problem could be constructed by connecting each node in the network once (and only once) with each node that it communicates. (This is not a complete cross bar which allows every node to communicate with every other node in one hop.)

The data presented here (for configurations allowing and refusing look-back) shows the communication radius for each benchmark. It is presented as a cumulative probability density, however there are some important differences between this measure (known as the *Cumulative Communication Radius Probability Density* or the *CCRPD*) and the CIPD discussed in section 5.2. If the CCRPD is represented by the function $\Phi(n)$, which is the probability that a given node will communicate with n or fewer unique nodes. Φ is cumulative to the left (whereas the CIPD is cumulative to the right).

Figures 6.11 and 6.12 show that for a particular node, a relatively small number of nodes will be communicated with. In fact, the largest such number is 9, showing that the 10-dimension hypercube could most likely be arranged to allow for ideal communication. In virtually every case, except for the Method of Moments, 50% of communication occurs between one pair of nodes, suggesting that the data placement in a ring could be optimized beyond what will be presented in section 6.9.1.

Allowing for look-back generally reduces the number of nodes which participate in the communication radius, however, not significantly (typically by 1-2 nodes). The benefit of look-back is primarily derived from increasing the run length on a

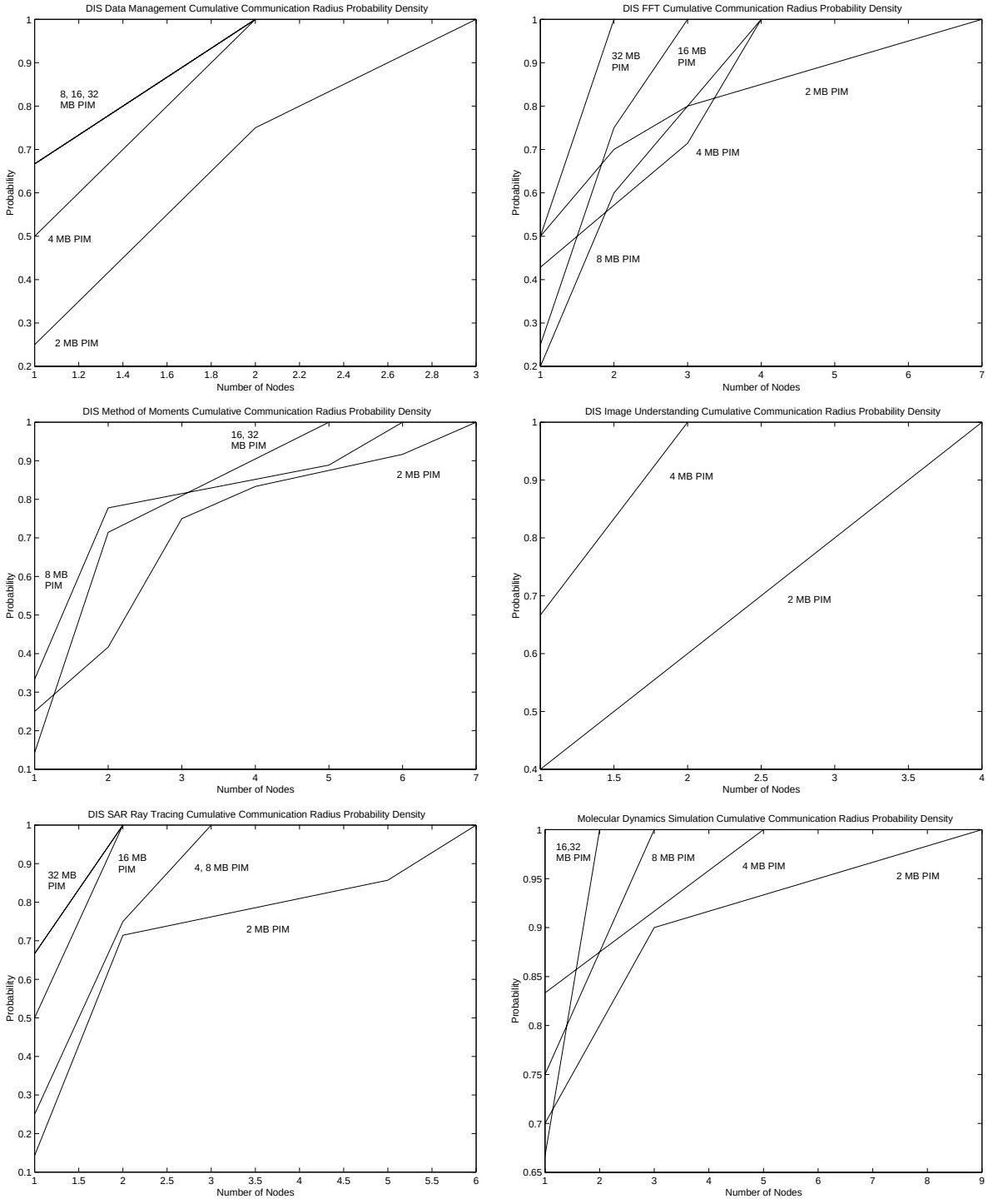


Figure 6.11. Cumulative Communication Radius Probability Density (without look-back)

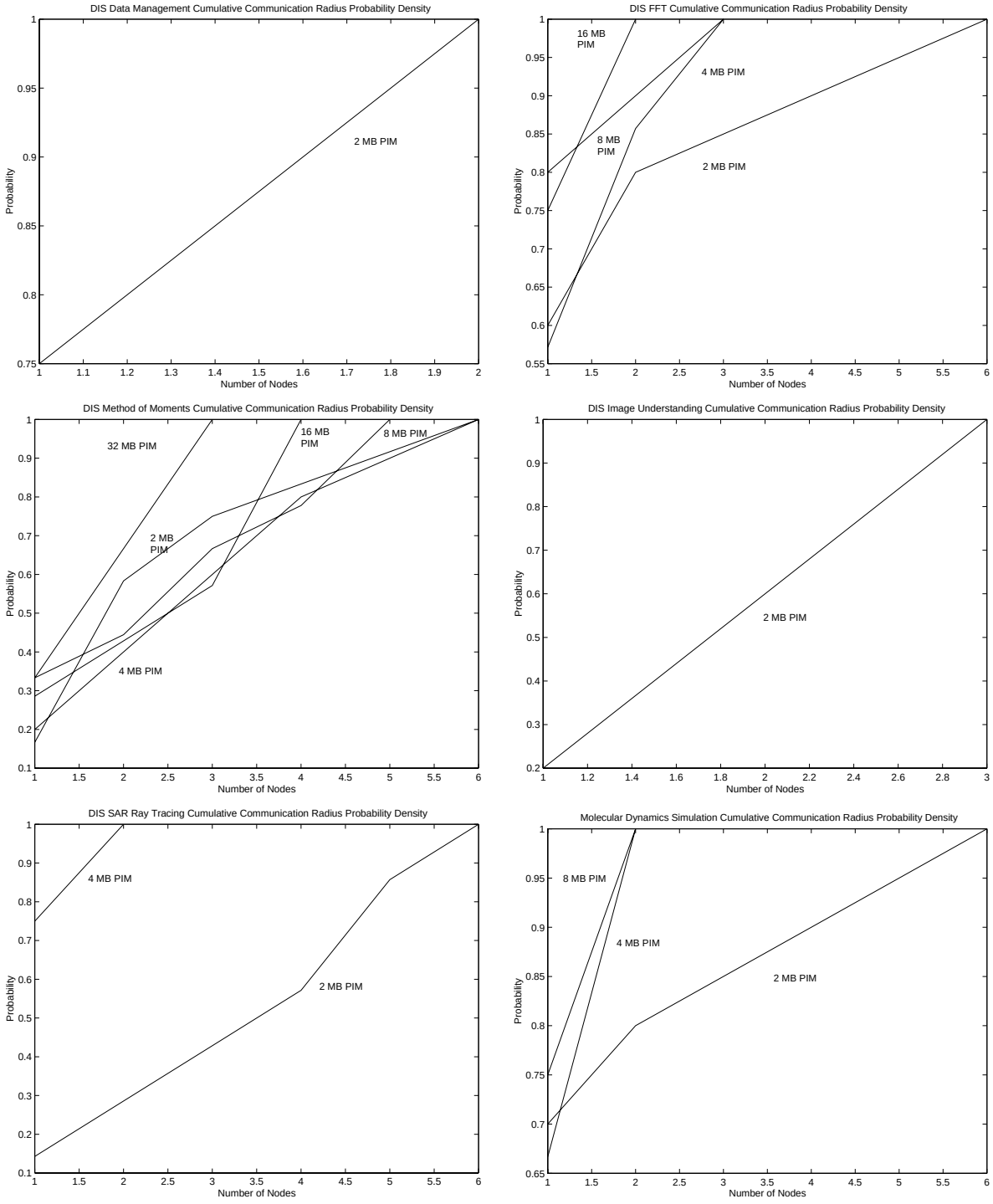


Figure 6.12. Cumulative Communication Radius Probability Density (with look-back)

given node.

Not surprisingly, increasing the size of each PIM's memory also significantly reduces the communication radius for a given node quite significantly. Some benchmarks, notably the Method of Moments and FFT, show a crossing of the CCRPDs when the configuration changes. This is easily accounted for by the fact that the data placement over a given size also changes significantly. Additionally, the lines which cross are always grouped close together indicating that the change is relatively insignificant.

The configurations which allow looking back (or packaging of data between thread movements) require a communication radius of 6 or less to satisfy every demand placed on the network. This indicates that a hierarchical network configuration, in which non-communicating PIMs are grouped far apart would allow for significant improvement over a network which is more uniform in structure.

6.8 Individual Network Performance

Now that the overall structure of communication is understood in a theoretical sense, the behavior of individual interconnection network implementations can be examined in detail. One final metric is required to do so successfully: the *Communication Probability Density*, or *CPD*, which represents a measure of the probability a given communication (caused by a memory miss) will result in a given number of hops or less. Since hops represent the distance between two nodes given a particular network topology, assuming an initial data placement, this is a measure of the success in placing data close together as well as a verification of the more abstract communication radius analysis presented above.

The CPD is represented by $\Upsilon(n)$. For a given number of hops, n, $\Upsilon(n)$ represents the probability that given a communication request n hops or fewer will be required.

As with the previous CCRPD ($\Phi(n)$), $\Upsilon(0) = 0$. Furthermore, if γ represents the largest distance between any two nodes on the network, then $\Upsilon(\gamma) = 1$ in all cases.

6.9 Memory Space

It is assumed that for each configuration, a 4 GB memory space is available (even though that space may not be entirely filled). Thus, the networks for the 2 MB PIM configuration had 2048 nodes, while the networks for the 4 MB configuration had only 1024 nodes. While the size of the network could have been optimized for the particular benchmark being run – after all, the smaller a ring is the lower the communication overhead – doing so would not represent a "real machine." Actual computers have fixed interconnection network topologies which cannot be dynamically reconfigured.

6.9.1 Ring

Although perhaps the simplest interconnection network studied, the ring generally produced a very high communication cost. While the ideal network would satisfy over half the requests with only one hop, even the best benchmark results on the ring (the Image Understanding) could not satisfy 40% of requests with one hop when look-back was prohibited. Figure 6.13 further shows that many of the benchmarks (the Data Management, Method of Moments, and Ray Tracing) required a very larger number of hops (a hundred or more) in many cases.

Allowing for look-back clearly improved the situation tremendously. In fact, Figure 6.14 shows that in some cases well over half the network requests could be serviced in one hop (see the Image Understanding and Method of Moments runs). Unfortunately, there were still many requests requiring hundreds of hops. The 2 MB configurations suffered disproportionately from this, whereas the larger configurations were more likely to be immune because they possessed a larger quantum

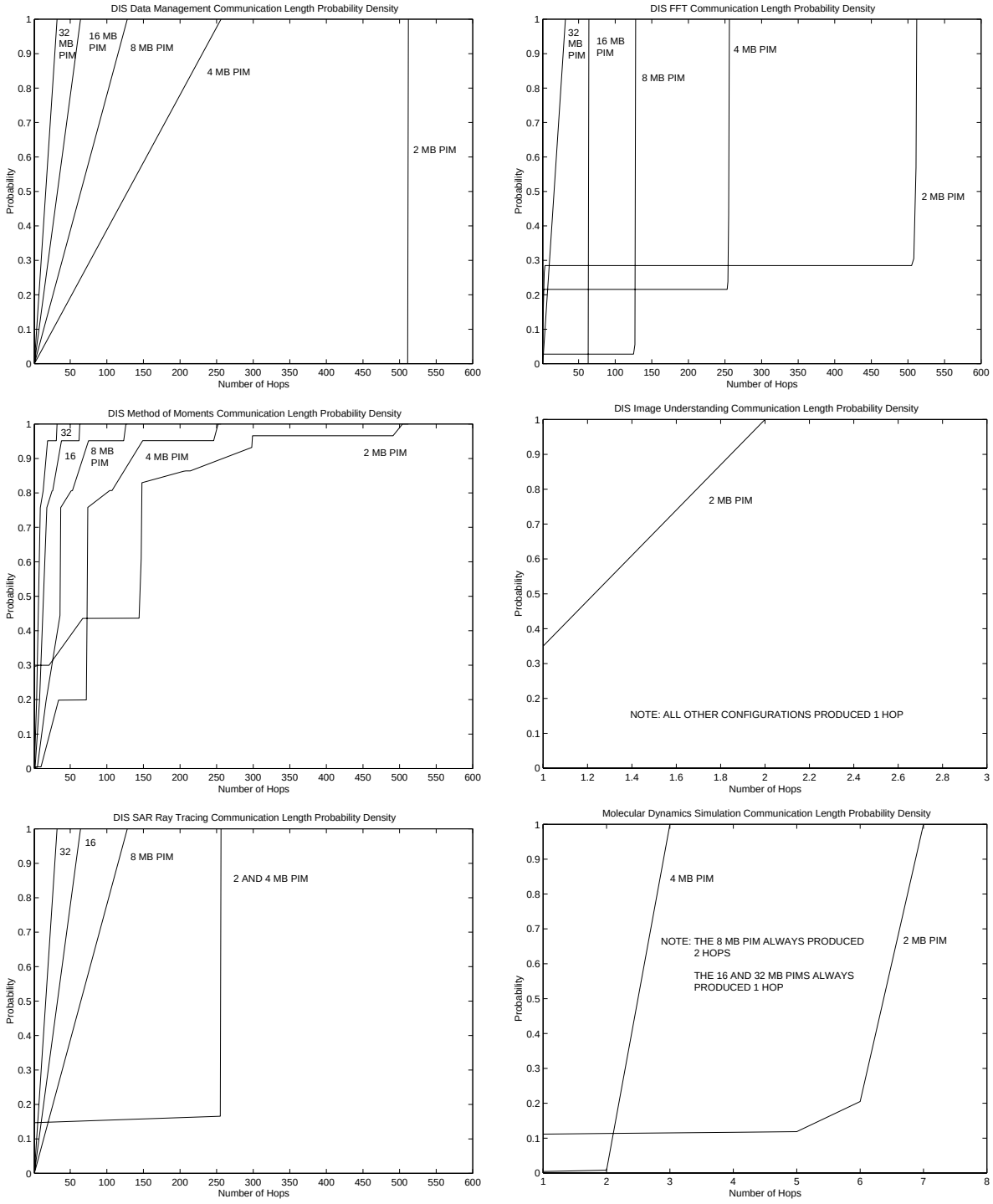


Figure 6.13. Ring CPD (without look-back)

of data.

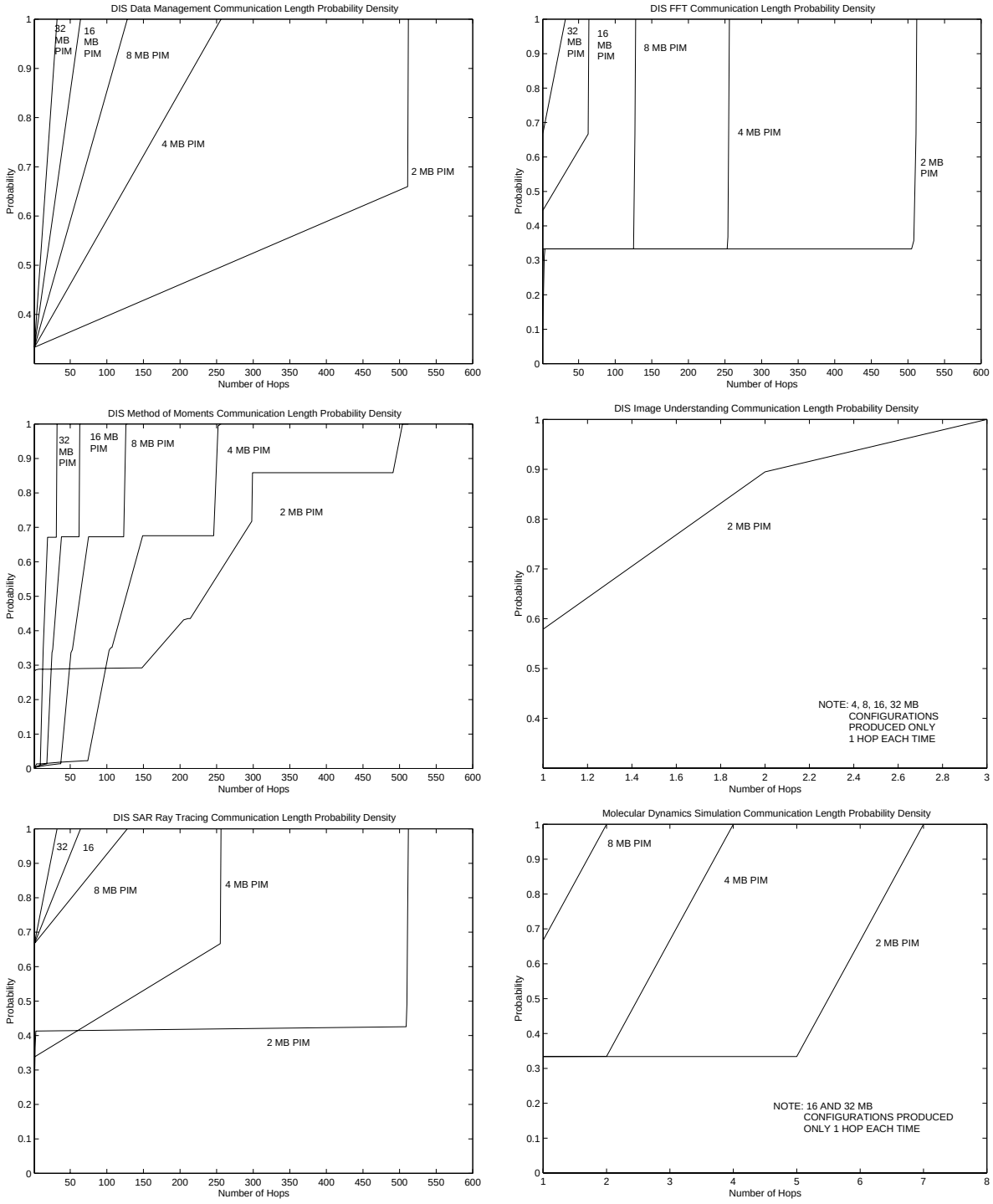


Figure 6.14. Ring CPD (with look-back)

6.9.2 Mesh

The mesh configurations proved better able to handle the load, as can be seen in figures 6.15 and 6.16. While the look-back feature did significantly improve the

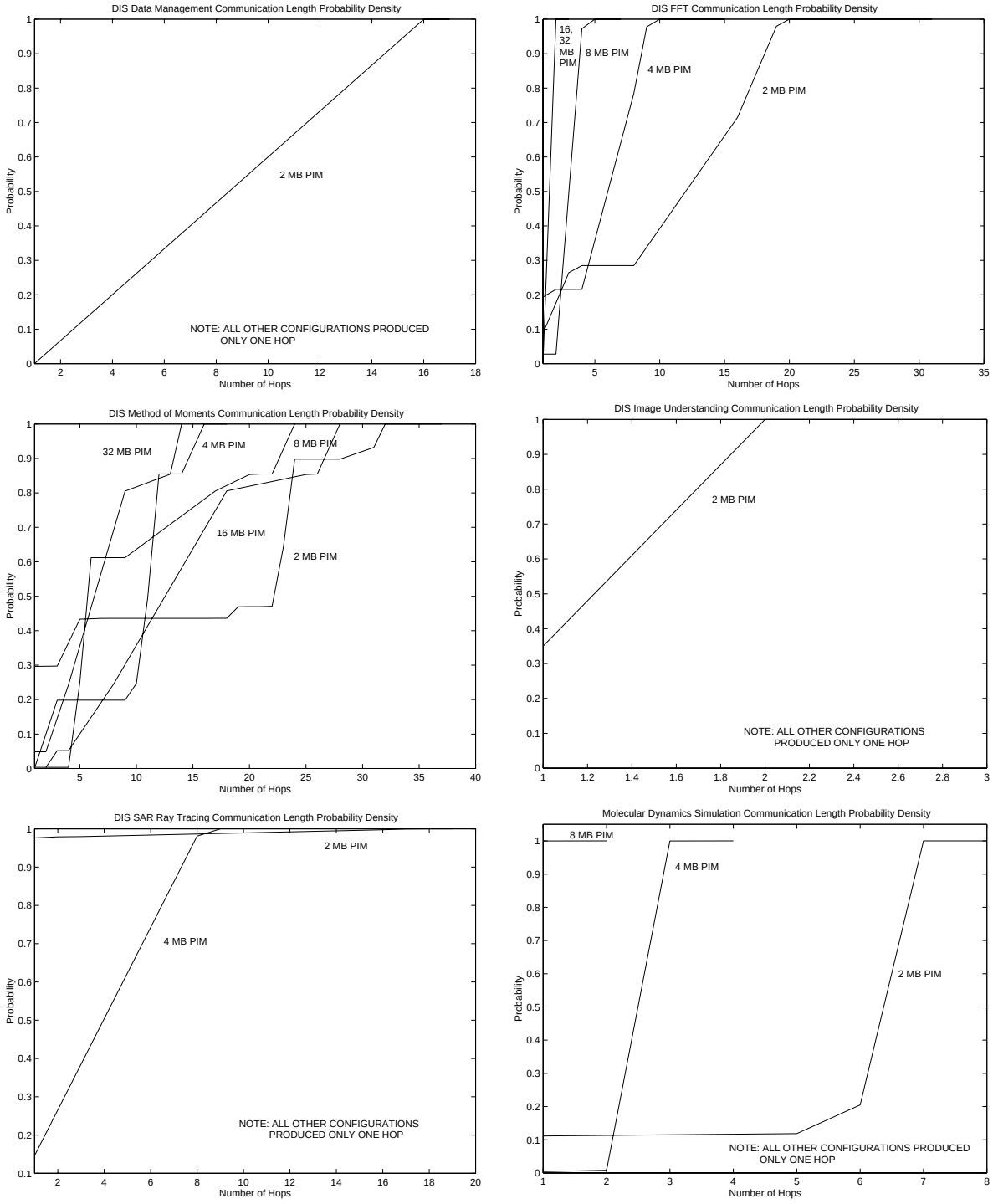


Figure 6.15. Mesh CPD (without look-back)

results, it was nowhere near as much as in the ring. No communication request was more than 35 hops away, indicating that nearly an order of magnitude improvement

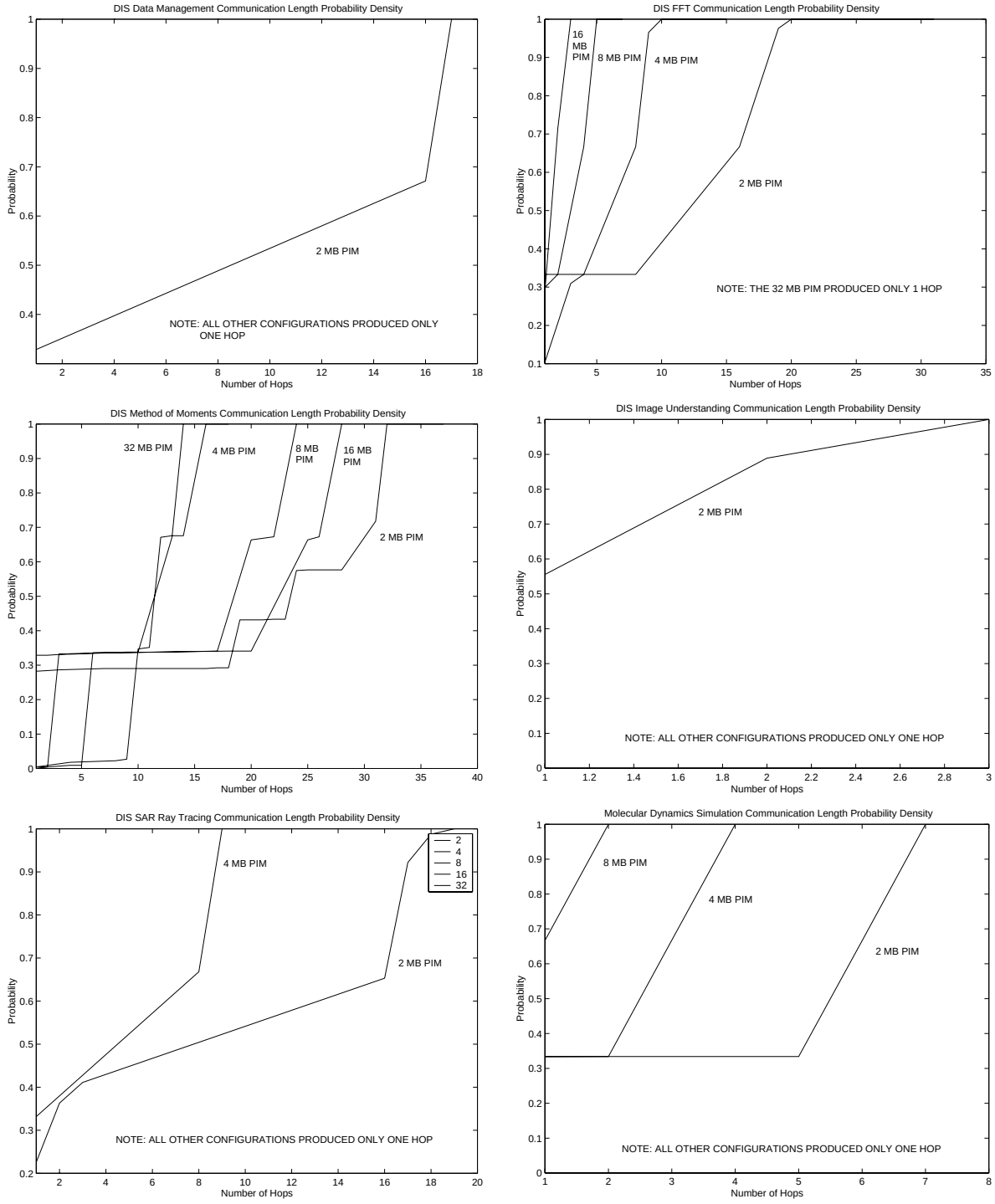


Figure 6.16. Mesh CPD (with look-back)

from the worst case in the ring can be seen. Additionally, the 50% points tended to be close to 5 hops, which indicates that half the requests could be serviced very

rapidly.

Allowing for look-back generally improved the results of the FFT, Image Understanding, Ray Tracing, and Molecular Dynamics Simulation. In each of the larger memory macro configurations, data locality improved and a greater number of communicating nodes were placed within one hop of each other.

6.9.3 Hypercube

Not surprisingly, the hypercube configuration showed the best results. Most communication, when look-back was allowed, occurred within 2-3 hops, with the Method of Moments benchmark again exhibiting the most demanding performance. Even without look-back, the hypercube managed to services most requests within 3-4 hops. Given the high bisection bandwidth of this type of network, along with the high connectivity, these results are will be expensive.

6.10 Communication Overhead

The communication overhead, in terms of the average number of hops required to reach a node capable of translating a given off chip address, is presented as a cumulative probability density in Figure 6.19. This is the mean for all PIM configurations. Since it is assumed that there are 4 nodes capable of translating addresses, all of which are evenly distributed around the interconnection network, the mean changed very little regardless of PIM size.

Changing the number of nodes will, of course, effect the outcome of these results. One node would increase the overhead (because, on average, a given node is farther away on the network), and fewer nodes will decrease the overhead. The 4 node configuration was chosen merely to demonstrate that the number of nodes performing a dictionary service does not have to be large.

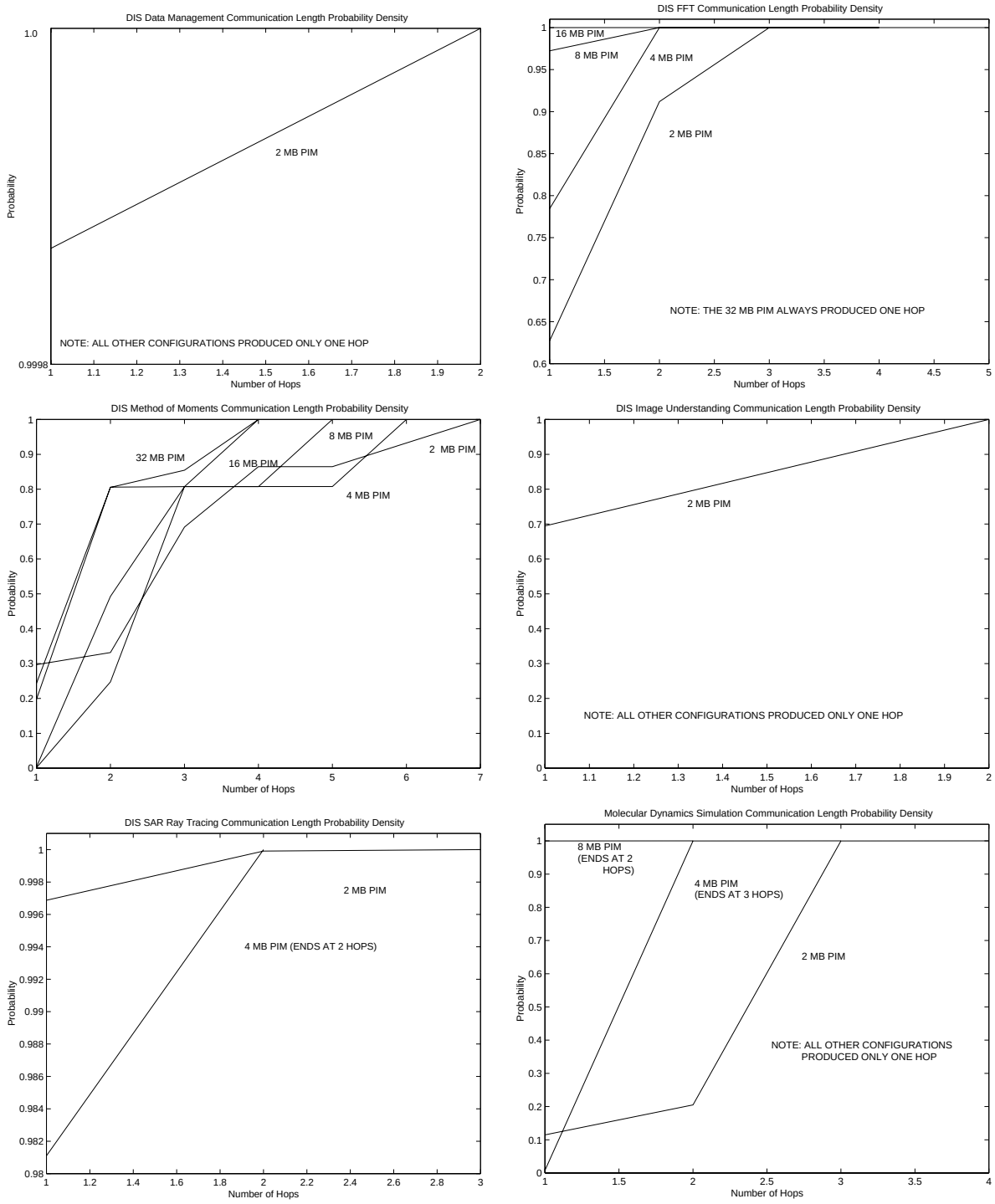


Figure 6.17. Hypercube CPD (without look-back)

6.11 Local Translation Mechanisms

Allowing for a simple *source translation* mechanism (as described in Section 6.3) is extremely beneficial. Figure 6.20 shows that remembering the physical location of

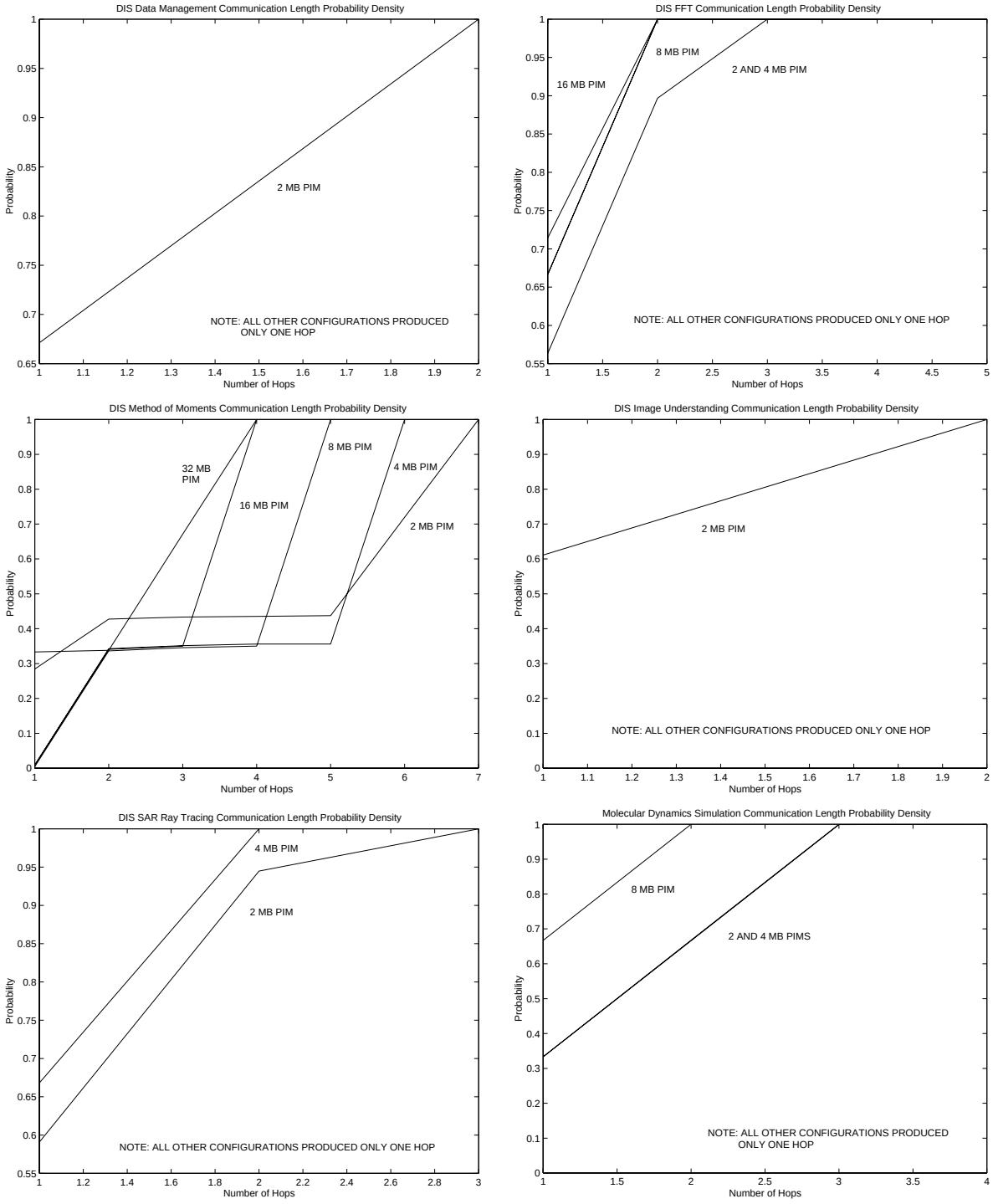


Figure 6.18. Hypercube CPD (with look-back)

the previous two remote memory references yields a miss rate of less than 10% for every benchmark studied.

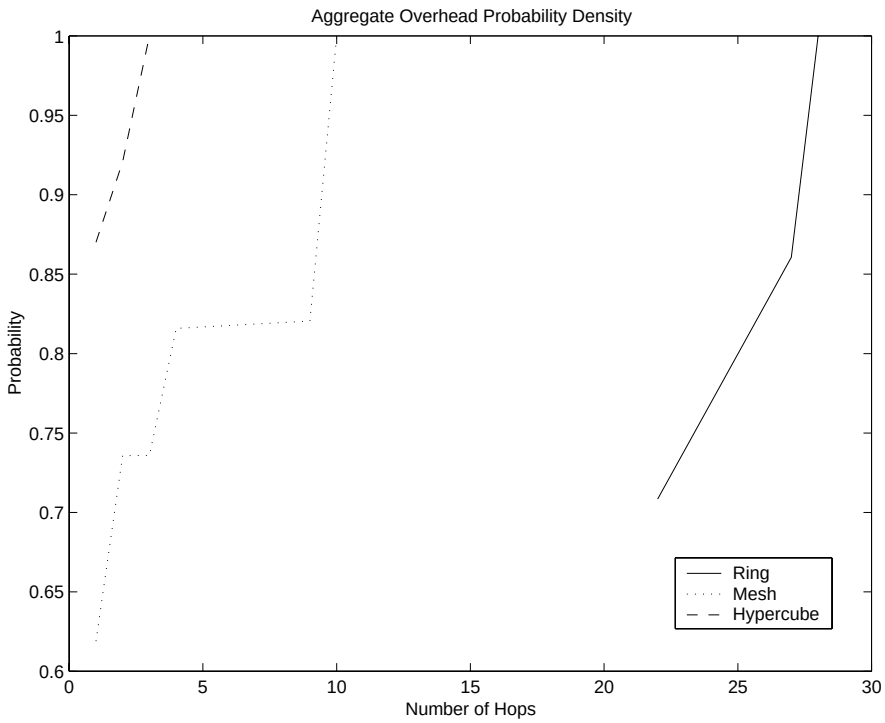


Figure 6.19. Aggregate Overhead

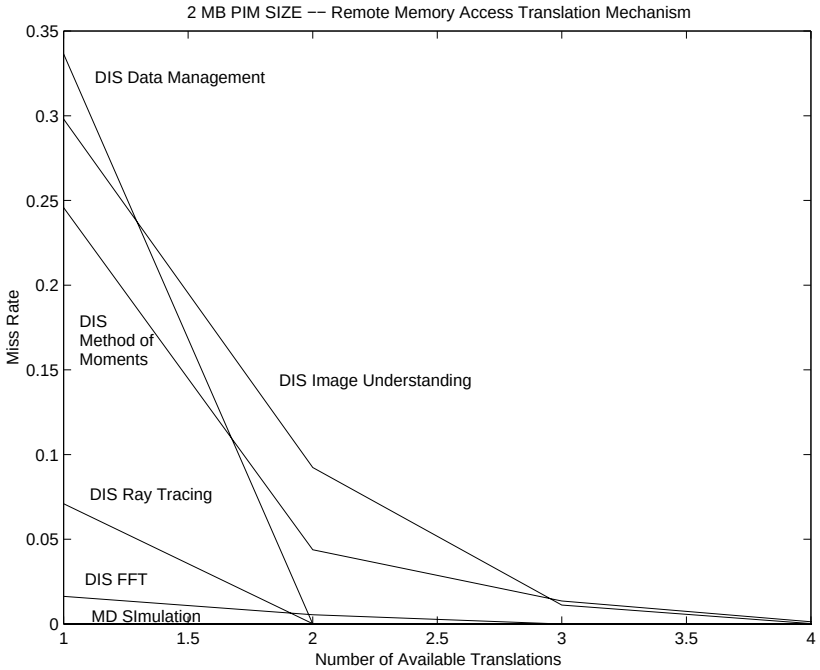


Figure 6.20. Remote Name Translation Mechanism Miss Rate (2 MB PIM)

Large PIM sizes significantly lower the miss rate of this translation mechanism. Figure 6.21 shows that for a 32 MB PIM, storing only one previous translation is quite effective. In fact, it reduces the miss rate for every benchmark except the Data Management and Method of Moments to virtually zero. Although the two graphs presented in this section represent (respectively) the worst and best case results, the unabridged data can be found in Appendix D.

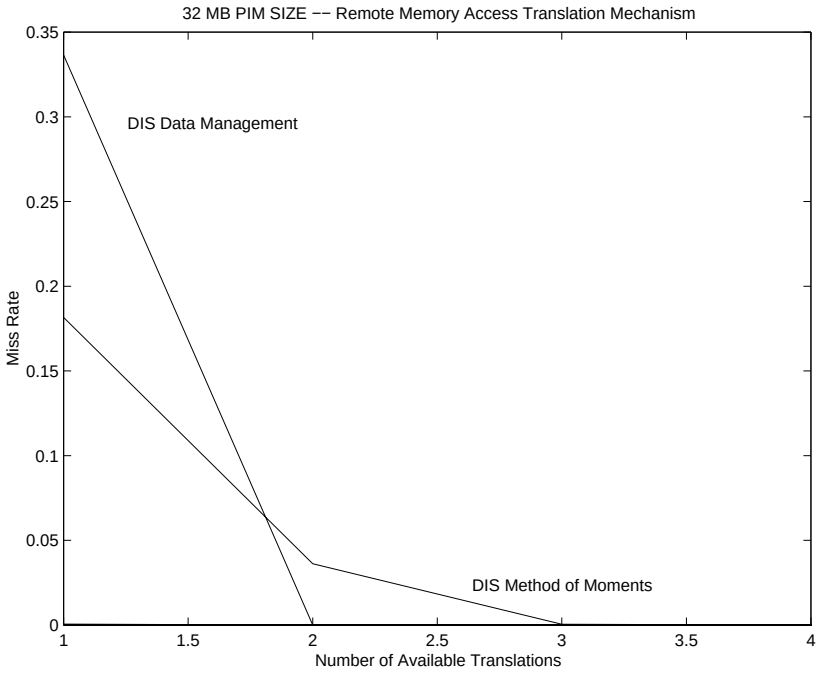


Figure 6.21. Remote Name Translation Mechanism Miss Rate (32 MB PIM)

6.12 Conclusions

Each benchmark has demonstrated itself to be well suited to a mobile thread environment. Because each of these benchmarks continuously requires new data to be made available (that is, the reuse is relatively low), long run lengths were obtained by moving from one node to the next while consuming as much local data as possible at each node. Clearly this is advantageous to a PIM system where the processing and

data being processed are tightly coupled. Furthermore, since each of these benchmarks are known to have good parallel implementations, the numbers presented in this chapter represent the absolute worst case. A mechanism for transferring the execution of a single thread from one node to the next could be implemented entirely in hardware without any support from the programmer or compiler whatsoever. Therefore, these lower bound numbers are extremely encouraging.

This simple data placement shows that the data structures that each of the benchmarks rely upon could be reasonably well placed throughout a PIM system by the runtime system without any compiler or user annotation. Multi-threaded implementations of each of these benchmarks can be developed on top of this type of simple data placement because of the latency tolerance inherent in multi-threading. The results show that mobile threads carrying a small amount of context with them could achieve significant run lengths on each node. Furthermore, this context can include a very small amount of data from the current node. This indicates that thrashing under such a model is relatively unlikely; thus, the communication overhead associated with moving a thread is amortized by the gain made by bringing it physically closer to the data being processed.

The overhead involved in terms of hops across the communication is relatively low. Since nodes tend to communicate with only a small number of other nodes, even a simplistic arrangement (in this case, placing sequential addresses close together) produces meaningful results. Additionally, the overhead required to determine upon which node a given address resides is relatively low provided that some nodes serve as a directory. A small number of nodes (in this case 4), when placed evenly throughout the system, prove to be easily capable of serving dictionary requests with low communication overhead.

Providing two remote memory address translations on each node (for each thread)

proves to be highly effective in speeding up the translation process. In fact, considering that the miss rate for this two translation TLB-like structure is nearly zero for every benchmark studied indicates that providing elaborate dictionary structures for remote name translation is unnecessary. When further coupled with the fact that PIMs can effectively take on very large chunks of the address space, these directory structures can be small. Additionally, since very few, small structures are needed, updating them will be relatively easy.

Finally, it should be noted that since each node held contiguous pieces of data (from 2 MB to 32 MB) in these experiments, only a small number of windows into the address space are needed. The results show that if a small amount of highly reused data is available, that, combined with the current segment being analyzed, allows for very long execution times between misses. This means that a very large page (or segment) size is feasible and even desirable. The advantage of using such large chunks of data is that translation information can be even further compacted (see Chapter 4). For example, if a node wishes to determine if it owns a given address it simply performs a single compare (or in the case of segmentation, one simple boundary check) can quickly determine if a given address should be sent off chip. Nodes which provide translation mechanisms can also store smaller tables and therefore perform faster searches. Finally, if the code is mobile, an intelligent run-time system could allow for a deterministic mapping of segments to nodes and support constructs for memory allocation which place associated pieces of data close together rather than "in the next free space."

CHAPTER 7

CODE, THREADS, CONTEXT, AND CARPET BAGS

> To build may have to be the slow and laborious task of years. To destroy can be the thoughtless act of a single day.
>
> – Sir Winston Churchill

In Chapter 6 it was shown that in a mobile thread environment, when a thread moves, but carries with it needed data from the previous node upon which the thread resided, the run length between off node accesses is greatly improved. Or, more succinctly, that mobile threads often look back at the previous node. Furthermore, the amount of data which a mobile thread accesses on the previous node is relatively small but tends to have a high degree of reuse. This begs the question: how can a thread capture this data before it moves without specific programmer intervention? Furthermore, there is more than data to consider when answering this question. The code and stack must also be accounted for.

Chapter 5 showed that the amount of data required by the code and stack is smaller than 4 KB when kept on a nodes. A mobile thread must be able to carry enough of its own code and stack with it to execute successfully when it moves – or, at least start execution successfully. Because this is a multi-threaded environment, some latency can be tolerated, however, if a mobile thread were to arrive on a node and be unable to execute even a single instruction that would be intolerable.

This chapter introduces the notion of a *Carpet Bag Cache*, which is a mobile cache and synchronization mechanism associated with a thread. It further shows

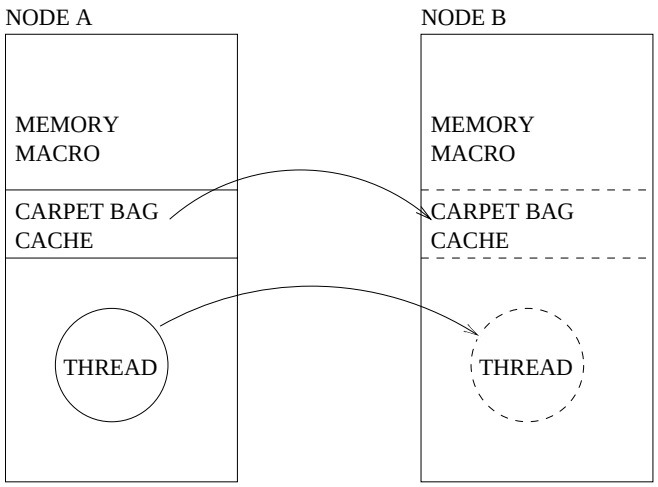


Figure 7.1. A Carpet Bag Cache moving from Node A to Node B

that the use of such a cache is highly successful even if the cache contains only a small amount of data. The success of the cache is measured in terms of overall miss rate of the cache and communication cost reduction.

7.1 Carpet Bag Caches

A *Carpet Bag Cache* (seen in Figure 7.1) is simply a small cache associated with a mobile thread which can move along with the thread. Its goal is to capture code, stack, and data references which will be needed by the thread after it has traveled. The code and stack references are of particular importance. It should be noted that on PIM the implementation of a carpet bag cache **might not** be done entirely in hardware. It may merely be a small buffer (in the memory macro) which contains the necessary data and is checked before an off node access is generated.

7.1.1 Implementation

There are two possible implementations for a Carpet Bag Cache. The most obvious implementation is as a standard general SRAM cache. This has the advantage of allowing a node to have access to a general purpose cache (see Figure 7.2).

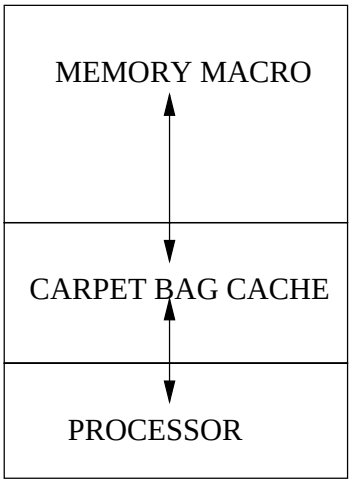


Figure 7.2. General Carpet Bag Cache

However, this may not be desirable. The memory latency on most PIM macros is quite low (even compared to the logic speed). Given that the machine is multi-threaded, tolerating this latency rather than using a complicated caching structure will be less expensive in terms of hardware cost, and it may significantly reduce the need for synchronization mechanisms. The alternative is to provide a small amount of support for the carpet bag cache in hardware and to use the runtime system to lower the latency of a potential off-chip access rather than eliminate it completely. This has the advantage of not only being simpler in hardware, but of being well suited to a multi-threaded environment.

Figure 7.3 shows a (primarily) software implementation of a carpet bag cache. The cache itself, which came from the previous node, is stored on the PIMs memory macro. When a program generates a memory access, the PIM determines if the requested data resides locally. If it does, the access can proceed normally, except that the address for that access is stored in a Memory Access Buffer (MAP), which is used to store the addresses of the most recently accessed wide words in memory. See Figure 7.3. However, if the access is a request for remote data the carpet bag cache

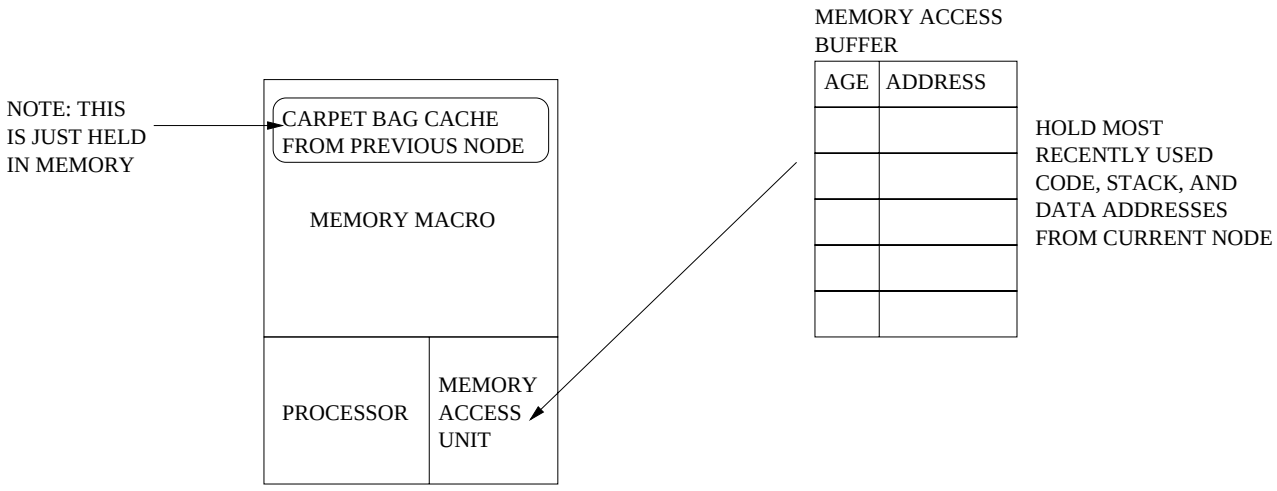


Figure 7.3. Software Implementation of a Carpet Bag Cache

must be searched to see if this "remote data" has, in fact, been imparted with the thread. This search can be done as a request for the cache from memory, followed by a small routine in software, or it could be done entirely in software by invoking an exception. If the data is found in the carpet bag cache, then the program can continue running. However, if it is not, the carpet bag and the recently touched local addresses can be merged to form a new carpet bag. Anything which does not fit can then be ejected back to the node to which it belongs. Figure 7.3 shows how this implementation would function.

It should be strongly noted that because this is a multi-threaded machine, and local memory accesses can be masked with a small number of threads, the high latency operations in the previous description (ie, the memory accesses) are presumed to be masked by other threads running on the node. The only true overhead involved is with whatever portion of the above described function is implemented in software. Because a remote memory access is such a high latency event, however, adding a small amount of code to search the carpet bag cache or package it up before delivery is significantly smaller than the communication cost.

Accesses to the code or stack should occur in a separate buffer during this configuration – specifically, a small number of wide words should always be available for reference. This is particularly important for the code.

7.1.2 Synchronization

Given that data no longer resides entirely on a single node, some simple form of synchronization must be maintained to prevent critical data from being corrupted. This is a necessary feature of multi-threaded systems regardless of the types of potential caching used. It is proposed that each wide word be annotated with a single "full or empty" bit similar to that employed in the Tera MTA. This status bit always begins set to empty. A wide word store to the given memory location will set the status bit to full. There are, then, potentially two types of load (for any part of the wide word): a *take*, which will set the status bit to empty and load the data, or a *copy* which loads the data without altering the status bit. In either case (*take* or *copy*), if the status bit indicates that the wide word is empty, the load will block until it becomes full.

These two types of load operation are very important given that not all data needs to be synchronized. For example, if self-modifying code is prohibited (virtually always a good idea), all instruction fetches could be copies. Furthermore, if a thread has its own stack (some threads may share a stack), there is no fear of interference from other threads. Allowing the programmer to decide seems entirely reasonable. However, given that one may wish to create an ISA independent carpet bag cache, the simple assumption that instruction fetches generate copies, and every other memory reference generates a take, is reasonable.

Finally, it should be noted that this mechanism may allow for *starvation*. If other threads depend on data that remains in a given thread's carpet bag cache,

they cannot continue their execution. There is no mechanism to guarantee that a thread will not hold onto such data for a very long time. In fact, *deadlock* could also occur if two threads become blocked waiting for data which the other has in its cache. Therefore some programmer awareness will be desirable.

7.2 Experimentation

The analysis program looked at both potential carpet bag cache configurations: first, the carpet bag cache as a general cache; and second, the carpet bag cache when used simply to reduce off-chip accesses. All the PIM sizes considered thus far (2, 4, 8, 16, and 32 MB) were examined with the same data distribution given in Chapter 6. The carpet bag cache itself a fully associative true LRU cache with 256 bit lines. All sizes from 1 to 1024 entries were examined.

Since the data distribution in Chapter 6 assumes that the address space is evenly divided into segments the size of a PIM, and an important function of this experiment is to determine the impact of code and stack references, all code or stack accesses are assumed to be off node. When a code or stack reference misses the carpet bag cache, it generates a remote memory request (to read the appropriate data), but the thread does not change nodes. This is meant to classify these accesses as the "worst case." In all likelihood, a large amount of code or stack data could reside on the node upon which the thread is currently executing.

Access to the data segment is treated differently. It is assumed that when a carpet bag cache miss is generated by a data access the thread should move to that data. This corresponds directly to the experiments in Chapter 6, except that the "look-back" mechanism is the finite size carpet bag cache (which has the added advantage of containing any data from any previous node which was not ejected by the LRU policy).

The Shade program used in this analysis tracks all loads, stores, and instruction fetches generated by the benchmark being analyzed. The effective address generated by each is then fed to the appropriate carpet bag cache. Each cache tracks the overall miss rate for a cache size of 1 to 1024 entries, since a true LRU cache of $N-1$ entries is always a subset of a true LRU cache of N entries. The statistics for these caches represent the case of a general cache.

Additionally, there are separate caches which track only off-chip references. A reference is considered "off-chip" if the node which it is attempting to access is different from the node which was used for the last access. Since multiple cache configurations can be run during each experiment, each cache data structure tracks the node used in the previous access.

7.3 Results

The carpet bag cache when used as either a general purpose cache or as merely a small buffer to hold references from the previous node proved extremely effective. Holding one wide word from the previous node cut remote accesses by 50% or more for every benchmark except the DIS Image Understanding.

Code and Stack references were also captured extremely efficiently. One wide word representing the stack generally satisfied most memory demands.

Surprisingly, the PIM size did not have any impact on the effectiveness of the carpet bag cache. The only exception to this is, again, the Image Understanding benchmark.

7.3.1 DIS Data Management

The simpler off-chip only configuration proved highly effective in capturing the "look-back" to the previous node. However, the cost was moderately high. Figure 7.4 shows that to capture 90% of the off chip accesses requires nearly 50 wide

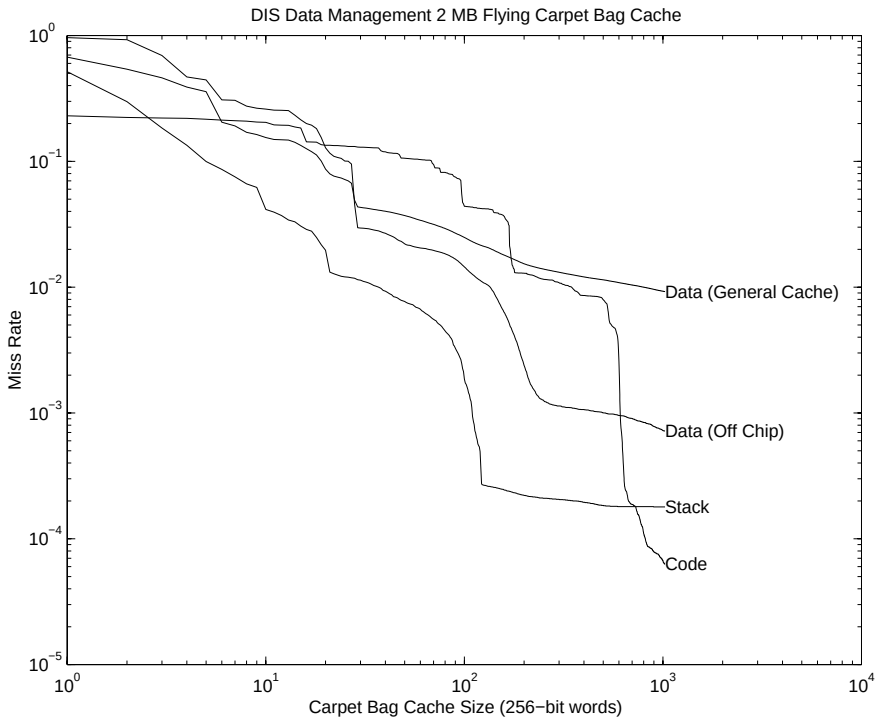


Figure 7.4. DIS Data Management Carpet Bag Cache Miss Rate

words.

Unsurprisingly, as a general purpose cache also proved highly effective. Roughly the same number of wide words achieved over a 90% hit rate.

Capturing the stack required nearly 10 wide words to achieve the same 90% hit rate. This is not surprising considering that the program is highly recursive. Although, it should be noted, that one wide accounted for around 50% of those hits.

The code was captured very effectively by one wide word, though, again, to achieve a 90% hit rate required a larger number.

7.3.2 DIS FFT

Figure 7.5 shows that one wide word again captured a very significant number of off-chip references. The off-chip data references fared exceptionally well. One wide

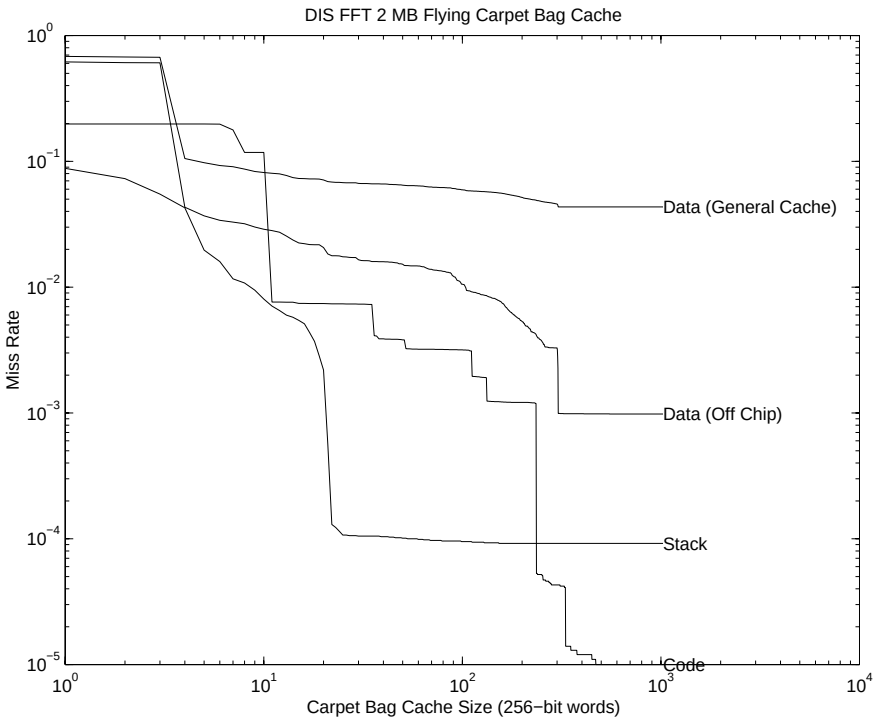


Figure 7.5. DIS FFT Carpet Bag Cache Miss Rate

word accounted for over 90% of them.

The code again required approximately 10 wide words to achieve the 90% hit rate, however, once again, the first wide word accounted for a very larger percentage of those hits.

As a general purpose cache, anything more than around 10 wide words was fairly ineffective.

The stack data was captured relatively quickly.

7.3.3 DIS Method of Moments

The Method of Moments data was captured very efficiently in the look-back-only configuration. As a general purpose cache, nearly 100 wide words (32 KB) were required to achieve a 90% hit rate. This vividly demonstrates that the benchmark exhibits low reuse. Furthermore, a 10 wide word buffer only eliminated 50% of the

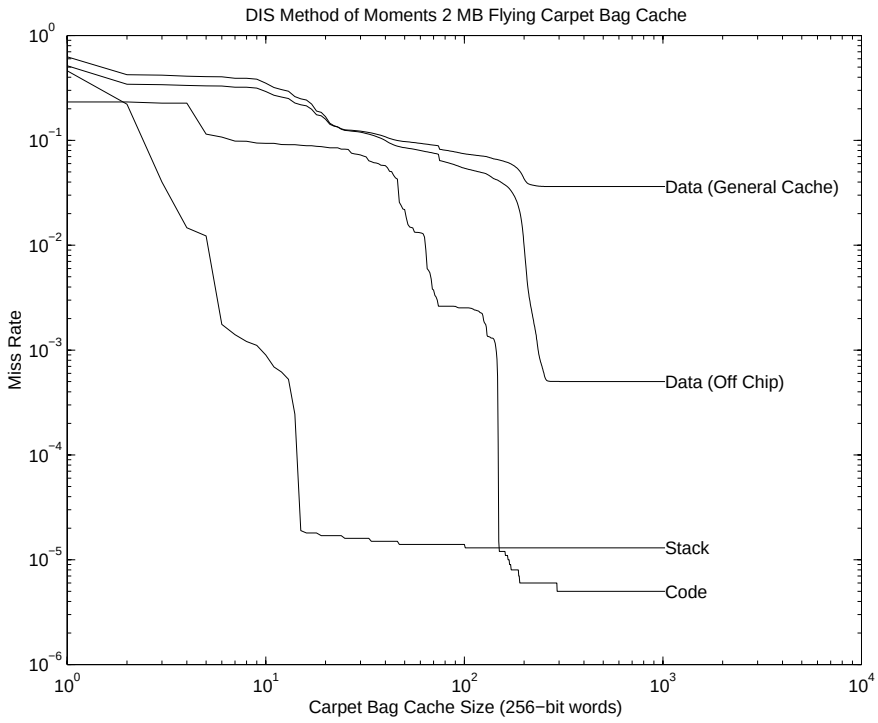


Figure 7.6. DIS Method of Moments Carpet Bag Cache Miss Rate

off chip accesses when used as a "look-back only" cache.

The stack and code references also required a relatively large number of words to achieve a high hit rate. See Figure 7.6.

7.3.4 DIS Image Understanding

The Image Understanding is the only benchmark which exhibited a change in the behavior of the carpet bag cache based on the size of the PIM node. Figure 7.7 shows the carpet bag cache miss rate for a 2 MB PIM, and Figure 7.8 shows the miss rate for a 32 MB PIM. (Complete results can be found in Appendix E.)

As the size of the PIM increased, the general purpose carpet bag cache did not change. However, the off-chip only carpet bag cache miss rate became significantly worse as the PIM size got larger.

The code data was captured extremely quickly compared to the other bench-

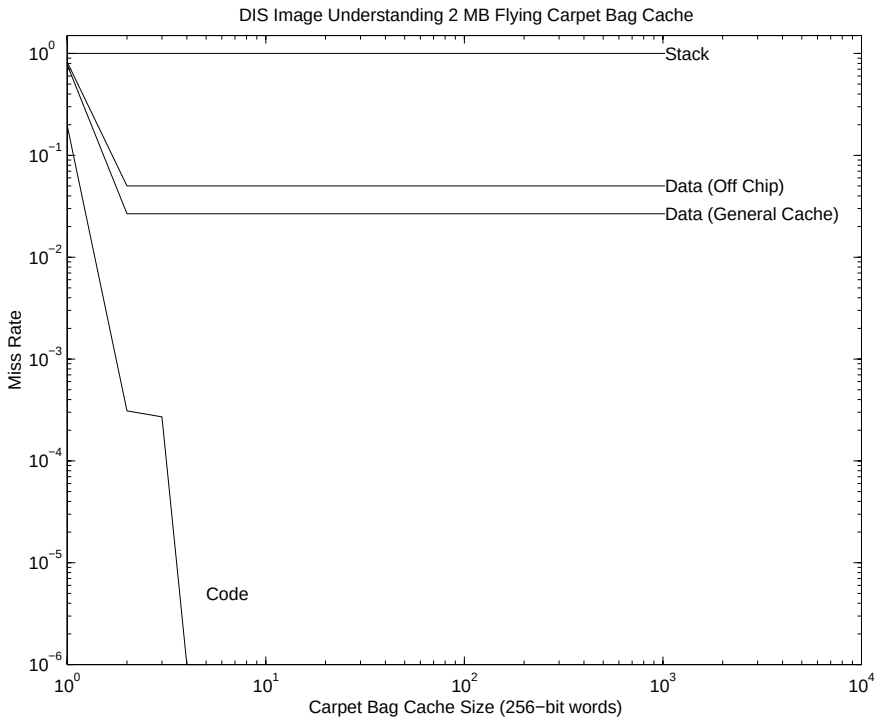


Figure 7.7. DIS Image Understanding Carpet Bag Cache Miss Rate (2 MB)

marks. Two wide words achieved a miss rate of only about 1%.

The stack also exhibits strange behavior. The miss rate is alway 100%. As Section 3.8 shows, the number of accesses to the stack throughout the benchmark is extremely low (on the order of 10 throughout the entire program), therefore it should be expected that misses will be frequently generated.

7.3.5 Molecular Dynamics Simulation

Not surprisingly, approximately 10 wide words (320 bytes) captured over 90% of all code accesses. The stack required only 5 wide words to capture almost all references.

The miss rates for the off-chip only and generalized data caches were virtually the same throughout the runs. See Figure 7.9.

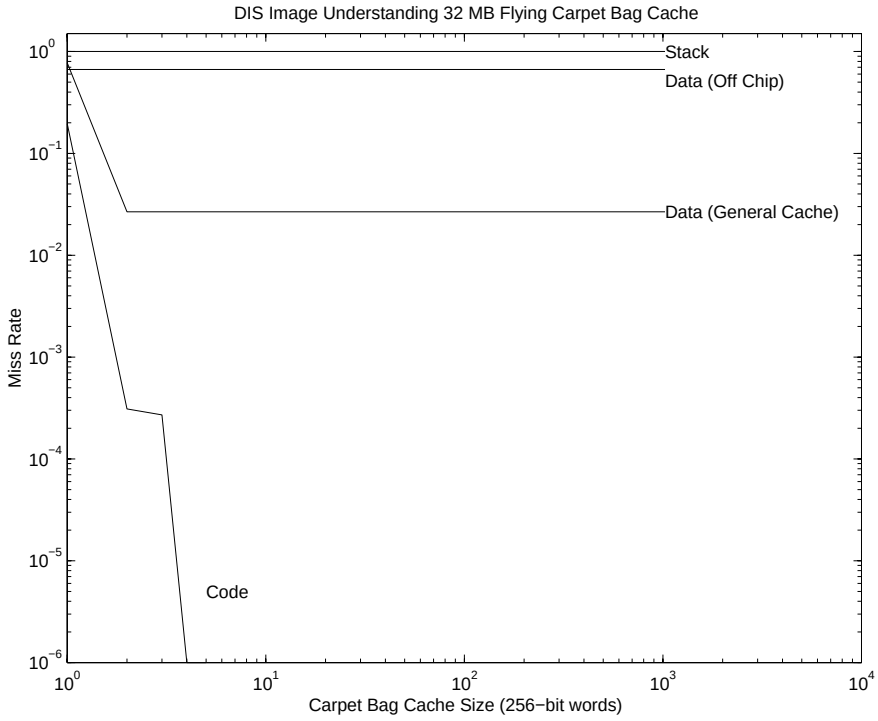


Figure 7.8. DIS Image Understanding Carpet Bag Cache Miss Rate (32 MB)

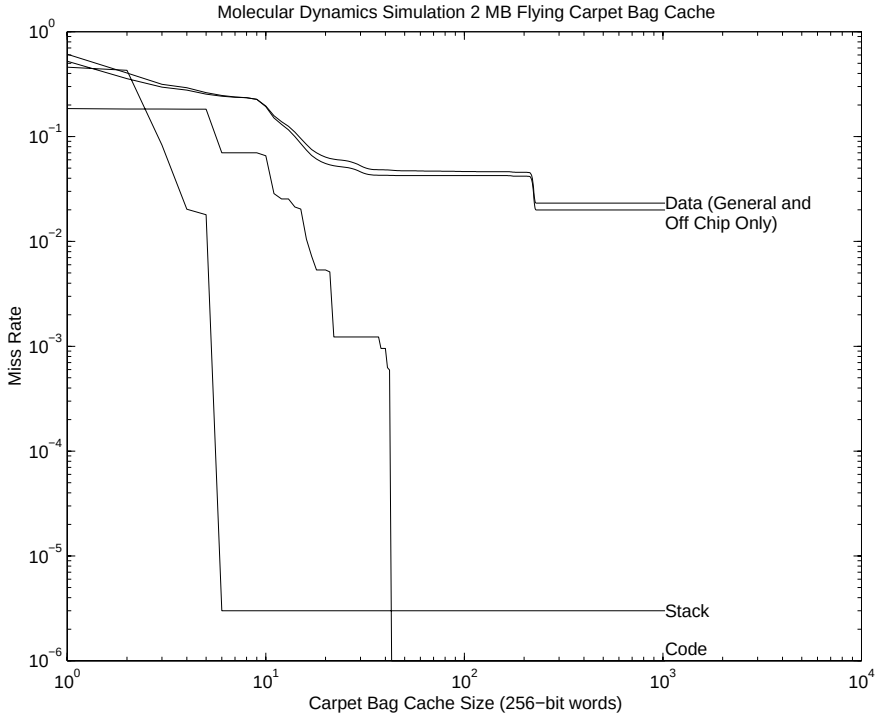


Figure 7.9. Molecular Dynamics Simulation Carpet Bag Cache Miss Rate

7.4 Conclusions

Instruction references are easily captured by 10 wide words. In general, the previously fetched wide word accounts for nearly half of all instruction references. In this case, the carpet bag cache is highly effective in capturing the code for a mobile thread.

Stack references generally required fewer wide words (5-7 to capture 90% of all accesses). The Image Understanding and Data Management benchmarks are the exception. In the case of the former, very few stack references are ever generated, so a caching mechanism will prove highly regardless of its size. In the case of the latter, a high degree of recursion makes the problem significantly more difficult.

However, a 50% success rate is virtually always achieved at the cost of only one wide word. Considering that this small price could potentially cut the communication costs in half, its significance cannot be overstated.

In terms of Data configurations, the "look-back" only (low-cost, software) configuration proved extremely effective. Not surprisingly, the general purpose cache required between 16 and 32 KB to be effective – this corresponds roughly to the size of a data cache on a modern workstation. On the other hand, storing only the off-chip references in a buffer eliminated a very large number of thread migrations at very low cost.

CHAPTER 8

CONCLUSIONS

This work examined the architectural parameters which effect program execution on PIM arrays using data intensive benchmarks. Because these benchmarks exhibit complex, non-uniform, memory request patterns, they proved to perfect test bed to flush out optimizations for the construction of PIM nodes which are highly tuned to take advantage of the low latency of on-node memory accesses. Furthermore, these goals were achieved at relatively low PIM resource cost, when compared to current parallel architectures.

The simulations executed constituted about 27,000 lines of code. Running the entire benchmark suite examined in this thesis (excluding Shade, *oo7*, and other runs used for verification) took a total of 1,173 hours.

8.1 Results

PIMs provide unique opportunities for low cost optimizations which would otherwise be difficult or impossible to incorporate into the architecture, such an ordering data to take advantage of the reuse of row buffers.

The paramount engineering problem, upon which this work was centered determining how much physical memory a PIM needs to sustain significant program execution; and, how the working set contained in that memory is best represented. Although larger memory macros always improved performance, a 2 MB PIM proved

sufficiently large to capture the working set and provide long periods of uninterrupted execution. Furthermore, using the memory to hold large pages, rather than as a cache, provided run lengths that were one to two orders of magnitude longer on average. The low overhead of this approach (in terms of the number of bits needed to represent the location of named data) makes it increasingly attractive.

Because large pages (256 KB and above, which provide few windows into the address space for a given node) proved so attractive during the working set experiments, simulation of a PIM array showed that this idea could be carried to the extreme. Managing a global address space in which each PIM is responsible for a single contiguous segment is extremely simple. The storage overhead (in terms of page tables for both local and global translations) is minimized. Furthermore, experimentation shows that individual nodes in the network communicate with a small number of other nodes. This means that tiling can be performed to minimize the overhead of communication by placing communicating nodes close together.

A simple execution model, involving the use of *mobile threads* was demonstrated to take full advantage of the large page size by transferring the execution of a thread from one node to another based on that thread's demands for data. Of course, when execution moves from one node to another, some data from the current node should be carried to the new node. Experimentation confirmed that the amount of data required is relatively small, on the order of hundreds of bytes.

Simulations conducted over a ring, mesh, and binary hypercube proved that the tiling need not be optimized by anything other than the order in which the programmer allocated data to be efficient. Furthermore, investing in more complex network structures (ie, the hypercube over the mesh over the ring) clearly provides tremendous benefit during this minimization.

Because any given node in the system will communicate with only a small number

of other nodes, resolving the physical location of any generated remote name can be done with a very small remote access TLB. Simulation demonstrated that the size of this TLB should be two large entries.

Finally, a simple caching mechanism, known as a *carpet bag cache*, was introduced to capture: (1) data from the current node which is needed when the thread moves to a new node; (2) the portion of the thread's code which must be carried around; and (3) the core of the thread's stack which also must be mobile. Two possible carpet bag cache implementations were examined. The most complex is a general purpose cache which can move from node to node, while the simpler implementation focuses on capturing only remote data. Both mechanisms proved highly effective in reducing the number of remote accesses, but given the significantly lower overhead involved in the largely software based remote data only caching scheme, combined with the already low latency of a local access, the simpler mechanism proved superior.

8.2 A Design Point

Figure 8.1 shows the configuration of a sample PIM node designed around the data derived in this thesis. The memory is organized into a large "global" area which represents the portion of the global memory space for which the PIM is responsible. This area holds large pages – potentially one large page. The local storage area, which cannot be addressed by by other nodes, is small and used to store temporary data (such as the contents of the thread's stack data, cached remote accesses, and remote address TLB). A small TLB is used to identify local accesses. Because there are large pages in the global page area and the TLB is enough to hold the translation for all of them, it will never miss. This serves to optimize the PIM for accessing local data.

Chapter 6 shows that very long run lengths can be achieved with a single large

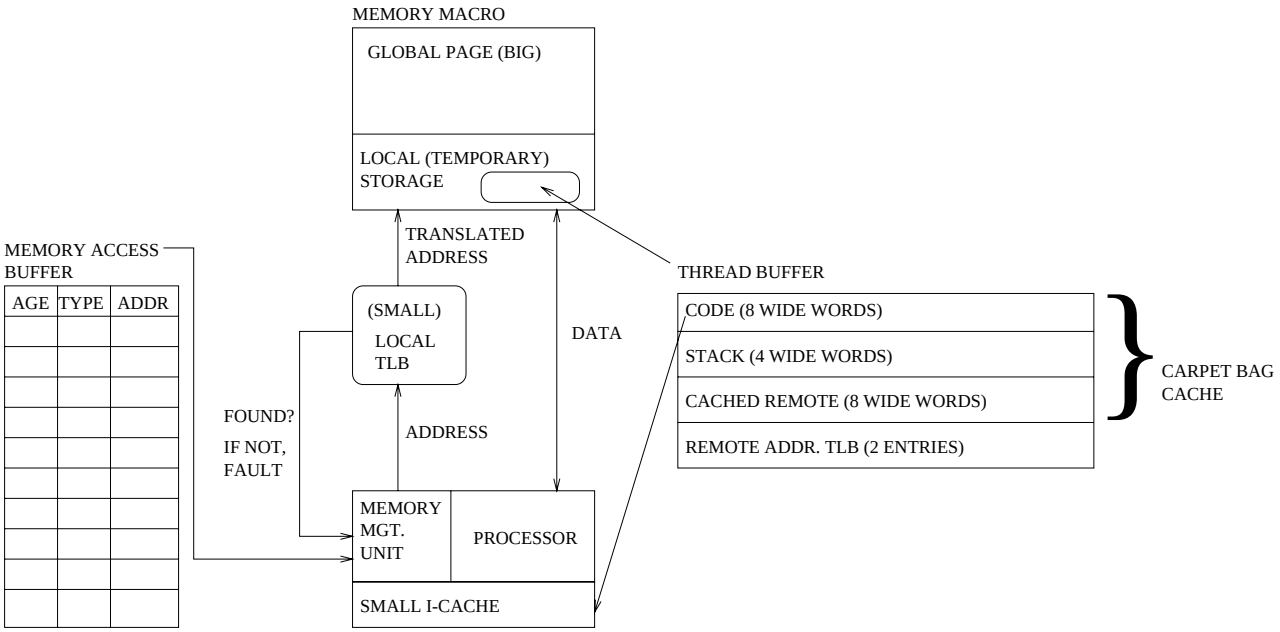


Figure 8.1. Proposed PIM Implementation

page on each node, however allowing for multiple pages would help to allow nodes to be a part of a computer.

8.2.1 Thread Buffers and Their Size

A *Thread Buffer*, which is merely a carpet bag cache with some history of remote address translation attached for routing the thread to its next node, is used to store the important data associated with each thread on the PIM. This buffer, primarily stored in the local storage area of the memory macro, can be backed up by small caches. Since threads can hide the latency of a local memory accesses, only an instruction buffer (or "cache") would be needed. This buffer can be very small (on the order of one wide word).

Simulation results indicate that approximately 8 wide words for code, 4 for the stack, and 8 for remote data accesses would easily satisfy at least 90% of remote memory requests. When these three cached types of information are combined

with a register file, a thread *context* is established for movement between nodes. This corresponds to a thread context size of 20 wide words, or 640 bytes, which represents **everything** that has to be moved when a thread changes location. This is a relatively small cost to pay over a modern interconnection network. (It is in fact less than 16% of a 4 KB page.) IN exchange for this relatively small cost, run lengths improve by one to two orders of magnitude.

However, there is a tremendous advantage in implementing the carpet bag cache in software (see Chapter 7) and storing the thread buffer in memory. Specifically, the size and contents of the thread buffer could be controlled by software (and would therefore be reconfigurable based on the demands of a given thread). Addresses stored in the Memory Access Buffer (MAB) can be annotated with a *type* field to indicate if they were instruction fetches, stack references, or data references (or, there could be three separate memory access buffers). Assuming the MAB stores N references of each type, the thread buffer could be configured to hold up to N references for each. Highly recursive threads, for example, could have their thread buffers reconfigured to hold a larger stack. Conversely, programs with high data demands could use their thread buffer's storage to hold a greater number of remotely cached wide words.

It is therefore recommended that the size of the MAB be at least 16 entries for code, 8 for the stack, and 16 for remote memory requests so that the system can provide maximum flexibility.

8.3 Global Page Area

As was previously mentioned, the Global Page Area should be as large as possible. However, experimentation in this work suggests that 2 MB is sufficient to hold a valid working set for most benchmarks. On the other hand, 8 MB is more than

sufficient for any of them.

8.4 Future Work

This work assumed the availability of mobile threads on PIM nodes. The next step is to validate the benchmarks in a true multi-threaded environment and examine how the run lengths compare to the results in this work. Significant improvement should be seen in applications which are thread aware. Determining the exact characteristics of how that multi-threading impacts the memory system is perhaps the most immediate outgrowth from this work. A simple run to completion thread package capable of interfacing with Shade simulations is currently under development. Rewriting the benchmarks to take advantage of multi-threading, and determining the exact impact of that design choice are clearly the next step in continuing this research.

This rewrite would further allow future simulations to account for traffic to the interconnection network as well as the memory macro. A true accounting of each node's utilization could also be made. When coupled with fabrication technology parameters, extremely accurate simulations (in terms of timing parameters) of real programs could be performed.

Additionally, developing a detached simulation of the Memory Access Buffer and assessing its impact on the over architecture of a PIM node (beyond its size) would provide valuable insight into the overall performance of a node.

Determining with absolute certainty the location of the Thread Buffer must be done. While the local access hit rate is very high, in the presented design point the thread buffer sits inside the memory macro. An off chip memory reference is interpreted as "not here, check the thread buffer and if its still not found move the thread." The process of checking the thread buffer requires a trap into some

handler code (user or OS). The overhead of the trap may be very low (since switching threads is assumed to cost no more than a jump), however, the actual code to search the buffer may produce significant impact (particularly for stack references). Determining the cost of that search by actually generating ASAP code to do it must be done before a final implementation can be decided upon.

Finally, all of the experiments in this work focused on PIMs executing a single program. Taking into account the effects of a multiprogrammed PIM system would be of great value.

The end result of this effort would describe an efficient, latency tolerant, and highly scalable PIM system architecture. Such a system would take full advantage of the high local bandwidth inherent in PIM designs and would be capable of extremely high throughput when compared to von Neumann machines.

APPENDIX A

BENCHMARK INSTRUCTION FREQUENCIES

Table A.1. DIS Data Management Instruction Frequency

Opcode	**Number Executed**	**Percent of Total**
lduw	1,823,758,820	20.3030
nop	1,358,021,525	15.1181
subcc	689,843,337	7.6797
stw	607,178,428	6.7594
ldf	516,894,948	5.7543
add	418,422,143	4.6581
or	362,854,498	4.0395
ba	355,021,409	3.9523
bne	322,093,820	3.5857
sethi	319,641,435	3.5584
fcmpes	211,585,442	2.3555
jmpl	203,061,080	2.2606
call	200,039,104	2.2269
fbg	171,881,802	1.9135
be	134,530,809	1.4977
restore	117,695,198	1.3102
save	117,695,198	1.3102
ldub	108,591,000	1.2089
sll	107,301,769	1.1945
stb	78,249,192	0.8711
sra	73,003,856	0.8127
be,a	57,417,793	0.6392
and	55,063,733	0.6130

Table A.1 (continued)

Opcode	Number Executed	Percent of Total
sub	42,599,327	0.4742
stf	38,981,397	0.4340
ble	38,853,637	0.4325
bl	33,547,045	0.3735
srl	32,323,512	0.3598
andcc	31,624,690	0.3521
bl,a	30,565,320	0.3403
fsubs	28,741,612	0.3200
fbule	26,347,615	0.2933
ldsb	26,049,924	0.2900
bge	25,242,728	0.2810
fbuge	20,923,496	0.2329
bne,a	19,373,176	0.2157
fmuls	19,232,412	0.2141
bneg	19,062,745	0.2122
fmovs	17,200,745	0.1915
lddf	11,986,823	0.1334
orcc	11,724,332	0.1305
bgu	10,649,006	0.1185
xor	10,096,584	0.1124
bcc	10,047,977	0.1119
addcc	8,547,664	0.0952
fstod	8,399,917	0.0935
bg	7,789,609	0.0867
fcmped	7,751,901	0.0863
mulscc	5,666,839	0.0631
bcs	5,127,316	0.0571
bleu	2,672,272	0.0297
ba,a	2,439,120	0.0272
fitod	2,195,312	0.0244
stfsr	2,024,290	0.0225
bge,a	1,499,277	0.0167
std	1,402,486	0.0156
fmuld	1,383,133	0.0154
ldfsr	1,330,732	0.0148
stdf	964,884	0.0107
faddd	836,337	0.0093
ble,a	676,314	0.0075
andn	672,227	0.0075
fcmpd	668,857	0.0074
fbne	665,369	0.0074

Table A.1 (continued)

Opcode	Number Executed	Percent of Total
fdtos	661,875	0.0074
fdivd	657,452	0.0073
ta	604,919	0.0067
fdtoi	603,999	0.0067
sth	527,679	0.0059
andncc	515,562	0.0057
bcc,a	353,524	0.0039
fnegs	320,691	0.0036
fbul	184,430	0.0021
rdy	171,891	0.0019
wry	171,891	0.0019
bcs,a	88,563	0.0010
bg,a	27,909	0.0003
lduh	21,917	0.0002
faligndata	16,759	0.0002
ldsh	7,857	0.0001
bpg,a,pt	7,403	0.0001
fsubd	7,003	0.0001
bple,a,pn	6,986	0.0001
TOTAL	8,982,726,833	100.0000

Table A.2. DIS FFT Instruction Frequency

Opcode	**Number Executed**	**Percent of Total**
lddf	1,688,226,044	18.3438
fmuld	1,190,479,683	12.9354
stdf	1,044,846,589	11.3530
faddd	1,034,885,966	11.2448
fsubd	1,000,675,338	10.8731
add	837,518,354	9.1003
sll	411,815,795	4.4747
or	254,070,109	2.7607
sethi	179,040,687	1.9454
subcc	157,356,132	1.7098
sub	147,078,143	1.5981
lduw	144,522,830	1.5703
smul	123,742,333	1.3446
jmpl	82,359,413	0.8949
fmovs	78,377,490	0.8516
bl	77,535,432	0.8425
nop	67,790,764	0.7366
stw	64,681,922	0.7028
call	61,933,811	0.6730
sra	61,017,877	0.6630
wry	40,750,493	0.4428
sdivcc	40,742,433	0.4427
bvc	40,736,063	0.4426
fnegs	34,249,390	0.3721
restore	30,545,961	0.3319
save	30,545,961	0.3319
fdivd	28,650,349	0.3113
fcmpd	28,650,279	0.3113
fsqrtd	28,650,228	0.3113
bge	21,890,759	0.2379
be	21,600,323	0.2347
bne	19,643,702	0.2134
andn	19,119,424	0.2077
addcc	19,112,047	0.2077
fitod	19,108,168	0.2076
ldf	19,106,876	0.2076
fbe,a	19,100,160	0.2075
fbe	9,550,076	0.1038
bg,a	5,821,489	0.0633
ble	5,342,695	0.0581

Table A.2 (continued)

Opcode	**Number Executed**	**Percent of Total**
be,a	2,824,912	0.0307
bl,a	1,902,418	0.0207
orcc	1,737,178	0.0189
ba	1,554,256	0.0169
bg	976,763	0.0106
ble,a	725,105	0.0079
srl	615,118	0.0067
bge,a	590,864	0.0064
and	521,732	0.0057
andcc	445,530	0.0048
bne,a	210,410	0.0023
bcc	57,468	0.0006
bcs	55,600	0.0006
bleu	35,938	0.0004
bcc,a	32,140	0.0003
std	16,042	0.0002
stf	9,879	0.0001
ldd	7,963	0.0001
ldsb	7,328	0.0001
bvs,a	6,323	0.0001
andncc	5,785	0.0001
bgu	5,583	0.0001
TOTAL	9,203,242,055	100.0000

Table A.3. DIS Method of Moments Instruction Frequency

Opcode	**Number Executed**	**Percent of Total**
lduw	9,731,390,348	17.9267
add	6,080,897,059	11.2019
or	4,882,324,368	8.9940
ldf	4,497,761,517	8.2855
sll	3,335,411,563	6.1443
sethi	3,053,604,033	5.6252
stw	2,168,994,501	3.9956
stf	2,016,558,664	3.7148
subcc	1,911,430,970	3.5211
mulscc	1,567,450,402	2.8875
nop	1,396,447,589	2.5725
fmuld	1,336,079,798	2.4613
lddf	1,221,285,624	2.2498
jmpl	989,332,870	1.8225
call	977,429,647	1.8006
faddd	973,614,067	1.7935
orcc	677,673,430	1.2484
bl	642,885,557	1.1843
bge	551,507,101	1.0160
srl	511,866,121	0.9429
bne	483,631,083	0.8909
addcc	462,063,060	0.8512
ble	461,240,923	0.8497
fsubd	432,908,578	0.7975
be	381,220,227	0.7023
andncc	364,610,589	0.6717
stdf	341,130,647	0.6284
sra	281,905,870	0.5193
rdy	280,679,270	0.5171
wry	280,679,270	0.5171
sub	224,235,223	0.4131
restore	212,069,733	0.3907
save	212,069,733	0.3907
bl,a	148,587,023	0.2737
fmovs	143,569,997	0.2645
bcs	133,670,036	0.2462
andn	105,755,818	0.1948
bg	89,411,681	0.1647
ba	86,071,863	0.1586
fnegs	81,824,077	0.1507

Table A.3 (continued)

Opcode	**Number Executed**	**Percent of Total**
std	79,312,326	0.1461
ldd	68,324,638	0.1259
be,a	67,295,413	0.1240
and	50,996,801	0.0939
bleu	47,898,572	0.0882
stb	43,023,202	0.0793
fabss	38,241,900	0.0704
bge,a	38,124,064	0.0702
ldub	32,041,303	0.0590
bg,a	18,032,911	0.0332
ldsb	15,792,707	0.0291
andcc	12,249,733	0.0226
bne,a	10,806,821	0.0199
bgu	4,552,418	0.0084
ble,a	3,538,784	0.0065
bcc	3,167,825	0.0058
xor	2,816,565	0.0052
lduh	2,299,171	0.0042
bpg,a,pt	2,070,327	0.0038
bple,a,pn	2,070,158	0.0038
fcmped	1,636,963	0.0030
sth	1,017,230	0.0019
stfsr	981,418	0.0018
fbul	958,661	0.0018
fdivd	842,974	0.0016
fitod	723,817	0.0013
ldfsr	653,378	0.0012
ldsh	511,100	0.0009
ba,a	477,264	0.0009
fdtoi	376,108	0.0007
bpos	363,845	0.0007
bpos,a	352,026	0.0006
fbule	351,373	0.0006
fcmpd	327,759	0.0006
fbne	327,722	0.0006
fbuge	326,976	0.0006
addc	125,223	0.0002
bcc,a	80,306	0.0001
bgu,a	77,387	0.0001
TOTAL	54,284,485,196	100.0000

Table A.4. DIS Image Understanding Instruction Frequency

Opcode	**Number Executed**	**Percent of Total**
add	7,887,502,617	16.5166
subcc	7,844,866,598	16.4273
ldsh	4,888,720,442	10.2371
sll	3,828,755,078	8.0175
ble	3,823,237,810	8.0059
lduh	3,248,945,809	6.8034
or	2,622,769,841	5.4921
be	2,201,865,154	4.6107
ldub	2,198,705,414	4.6041
bne	1,720,496,949	3.6028
addcc	1,688,317,218	3.5354
sth	1,628,104,413	3.4093
sra	1,624,052,220	3.4008
bl,a	1,623,312,181	3.3992
lduw	156,659,421	0.3280
ldf	116,441,809	0.2438
sethi	91,907,036	0.1925
stw	70,431,448	0.1475
bl	68,691,571	0.1438
sub	66,140,907	0.1385
bg	63,086,929	0.1321
lddf	58,575,275	0.1227
nop	58,463,713	0.1224
fbule	57,671,936	0.1208
fcmpes	57,670,801	0.1208
bne,a	19,906,285	0.0417
mulscc	5,419,609	0.0113
be,a	5,124,671	0.0107
bge	4,071,407	0.0085
srl	3,473,737	0.0073
call	2,227,312	0.0047
jmpl	2,205,987	0.0046
orcc	1,823,370	0.0038
bgu	1,634,244	0.0034
bleu,a	1,622,593	0.0034
andncc	1,242,717	0.0026
restore	803,678	0.0017
save	803,678	0.0017
faligndata	801,774	0.0017
ba	737,057	0.0015

Table A.4 (continued)

Opcode	**Number Executed**	**Percent of Total**
faddd	637,992	0.0013
bg,a	635,040	0.0013
stf	539,520	0.0011
stdf	483,241	0.0010
fmuld	481,417	0.0010
rdy	448,193	0.0009
wry	448,193	0.0009
fitod	374,821	0.0008
bleu	340,697	0.0007
fadds	275,172	0.0006
andn	269,679	0.0006
bcs	251,273	0.0005
and	230,399	0.0005
fitos	224,128	0.0005
ble,a	214,987	0.0005
std	201,796	0.0004
fsubd	187,370	0.0004
ldd	149,552	0.0003
xor	114,204	0.0002
bge,a	114,117	0.0002
fcmpd	113,428	0.0002
fdtoi	112,655	0.0002
fble	112,064	0.0002
fstod	111,852	0.0002
bpe,pn	99,515	0.0002
lddfa	97,674	0.0002
stdfa	97,527	0.0002
andcc	75,027	0.0002
stb	70,539	0.0001
fdivd	39,497	0.0001
fdtos	37,495	0.0001
ldsb	35,124	0.0001
bpa,a,pt	32,050	0.0001
bcc	28,475	0.0001
TOTAL	47,755,056,628	100.0000

Table A.5. DIS SAR Ray Tracing Instruction Frequency

Opcode	**Number Executed**	**Percent of Total**
lduw	46,243,892	12.6216
subcc	44,528,100	12.1533
stw	37,447,853	10.2209
add	34,029,948	9.2880
or	24,000,280	6.5506
sethi	19,041,814	5.1972
be	14,275,074	3.8962
jmpl	12,710,449	3.4691
ldub	12,006,596	3.2770
call	9,656,516	2.6356
sub	8,570,149	2.3391
nop	8,478,687	2.3141
andcc	7,803,957	2.1300
bne	6,482,742	1.7694
save	6,316,891	1.7241
restore	6,316,890	1.7241
ba	5,277,937	1.4405
ldf	5,202,178	1.4199
sll	5,028,522	1.3725
be,a	4,958,660	1.3534
stb	4,217,232	1.1510
bge	4,127,542	1.1266
bl	3,319,375	0.9060
bneg	3,231,544	0.8820
bgu	2,685,166	0.7329
ldsb	2,571,200	0.7018
bne,a	2,453,264	0.6696
stf	2,263,505	0.6178
bg	1,781,428	0.4862
ble	1,617,015	0.4413
lddf	1,557,004	0.4250
sra	1,209,887	0.3302
bl,a	1,134,890	0.3098
orcc	1,043,893	0.2849
fcmpes	1,013,117	0.2765
bge,a	839,263	0.2291
fstod	827,237	0.2258
fdtos	728,069	0.1987
stdf	620,816	0.1694
stfsr	619,573	0.1691

Table A.5 (continued)

Opcode	Number Executed	Percent of Total
fbuge	619,553	0.1691
fcmped	573,928	0.1566
fsubs	508,374	0.1388
fbule	507,047	0.1384
fdivs	506,519	0.1382
and	482,634	0.1317
fcmpd	459,933	0.1255
fbne	459,806	0.1255
srl	418,487	0.1142
std	415,195	0.1133
fmuld	413,229	0.1128
fsubd	413,054	0.1127
ldfsr	413,042	0.1127
bg,a	362,227	0.0989
fbg	354,530	0.0968
bcs	232,979	0.0636
andn	209,398	0.0572
xor	208,434	0.0569
fmovs	207,643	0.0567
bcc,a	207,126	0.0565
ble,a	207,023	0.0565
bcs,a	206,646	0.0564
faddd	206,629	0.0564
fdivd	206,493	0.0564
fitod	206,406	0.0563
fabss	206,400	0.0563
sllx	157,259	0.0429
fbl	105,228	0.0287
addcc	91,792	0.0251
fnegs	88,775	0.0242
bpos	78,758	0.0215
addc	78,635	0.0215
movrlz	78,629	0.0215
mulx	78,629	0.0215
srax	78,629	0.0215
bcc	76,759	0.0210
bleu	3,840	0.0010
lduh	3,211	0.0009
mulscc	2,857	0.0008
fmuls	2,412	0.0007
ta	1,388	0.0004

Table A.5 (continued)

Opcode	**Number Executed**	**Percent of Total**
fadds	1,209	0.0003
ba,a	1,160	0.0003
bpg,a,pt	782	0.0002
bple,a,pn	767	0.0002
fbul	693	0.0002
andncc	658	0.0002
sth	399	0.0001
ldd	341	0.0001
bgu,a	314	0.0001
rdy	253	0.0001
wry	253	0.0001
ldsh	250	0.0001
TOTAL	366,385,719	100.0000

Table A.6. Molecular Dynamics Simulation Instruction Frequency

Opcode	**Number Executed**	**Percent of Total**
lddf	530,476,356	28.1821
add	273,367,562	14.5229
fmuld	228,411,606	12.1346
fsubd	154,799,406	8.2239
faddd	154,338,946	8.1994
stdf	69,100,689	3.6710
subcc	64,419,300	3.4223
bl,a	47,572,647	2.5273
fcmped	45,619,659	2.4236
nop	43,623,831	2.3176
fbg,a	40,312,500	2.1416
or	32,950,788	1.7505
fmovs	32,881,218	1.7468
lduw	32,872,462	1.7464
sethi	20,644,107	1.0967
stw	16,792,652	0.8921
sll	9,411,812	0.5000
call	8,163,767	0.4337
jmpl	8,040,871	0.4272
sub	7,319,085	0.3888
fdivd	6,807,119	0.3616
fbule,a	5,307,091	0.2819
be	3,207,566	0.1704
restore	2,690,005	0.1429
save	2,690,005	0.1429
andn	2,252,666	0.1197
bl	2,145,484	0.1140
be,a	2,099,689	0.1115
and	1,994,345	0.1060
ba	1,899,267	0.1009
bge	1,802,616	0.0958
bg	1,750,881	0.0930
sra	1,594,185	0.0847
bne	1,548,872	0.0823
std	1,527,922	0.0812
smul	1,421,883	0.0755
stf	1,344,809	0.0714
andcc	1,253,704	0.0666
ble,a	1,253,311	0.0666
fitod	1,226,676	0.0652

Table A.6 (continued)

Opcode	**Number Executed**	**Percent of Total**
orcc	1,169,555	0.0621
srl	1,074,373	0.0571
ldf	1,015,112	0.0539
bg,a	922,589	0.0490
faligndata	797,410	0.0424
ldub	754,261	0.0401
stb	751,615	0.0399
wry	687,760	0.0365
sdivcc	687,504	0.0365
ldd	656,454	0.0349
fdtoi	609,884	0.0324
ble	584,723	0.0311
bge,a	556,951	0.0296
bne,a	505,627	0.0269
bpe,pn	422,139	0.0224
bcc	408,816	0.0217
bleu	385,261	0.0205
bpg,pn	375,271	0.0199
bvs,a	375,004	0.0199
bcs	343,495	0.0182
andncc	338,269	0.0180
bvc	312,500	0.0166
fabss	290,962	0.0155
fnegs	261,981	0.0139
fsqrtd	250,001	0.0133
bcc,a	141,194	0.0075
bgu	110,958	0.0059
bpe,pt	93,796	0.0050
wrfprs	93,796	0.0050
bcs,a	86,038	0.0046
bpg,a,pt	62,560	0.0033
bgu,a	46,919	0.0025
alignaddr	46,898	0.0025
bpe,a,pn	46,898	0.0025
rdfprs	46,898	0.0025
bple,a,pn	15,662	0.0008
subc	15,624	0.0008
addcc	11,982	0.0006
ldsb	7,067	0.0004
ta	5,652	0.0003
mulscc	2,884	0.0002

Table A.6 (continued)

Opcode	**Number Executed**	**Percent of Total**
xor	2,102	0.0001
ba,a	1,177	0.0001
lduh	1,035	0.0001
TOTAL	1,882,315,390	100.0000

APPENDIX B

UNABRIDGED WORKING SET CIPD ($\Psi(L)$) RESULTS

B.1 DIS Data Management

B.1.1 Page Configurations

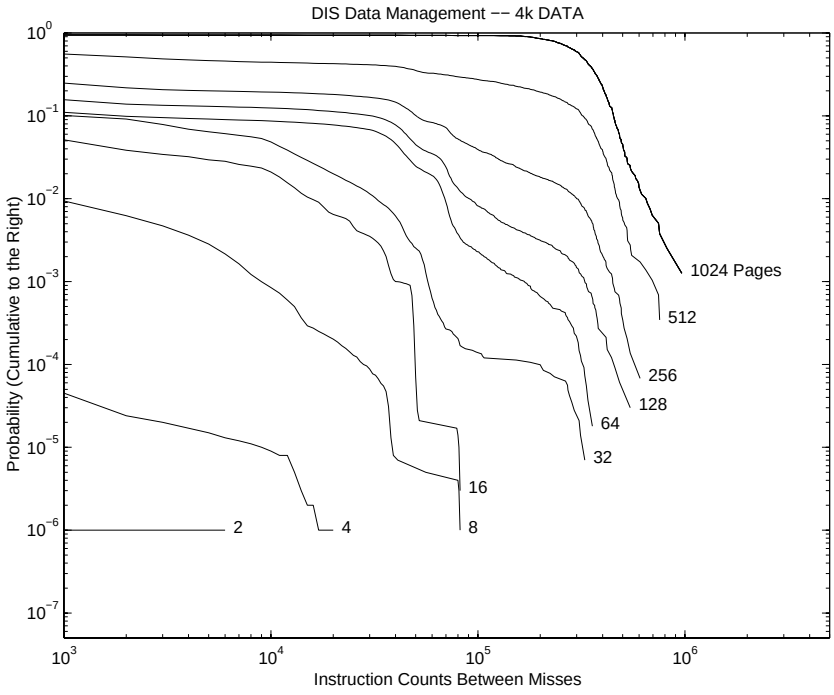


Figure B.1. DIS Data Management 4 KB Page CIPD (Ψ)

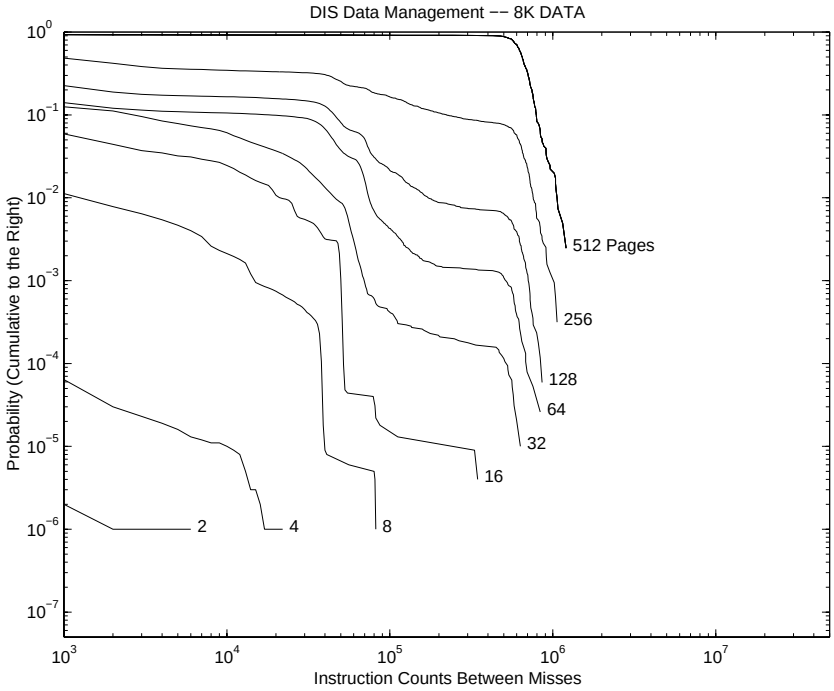


Figure B.2. DIS Data Management 8 KB Page CIPD (Ψ)

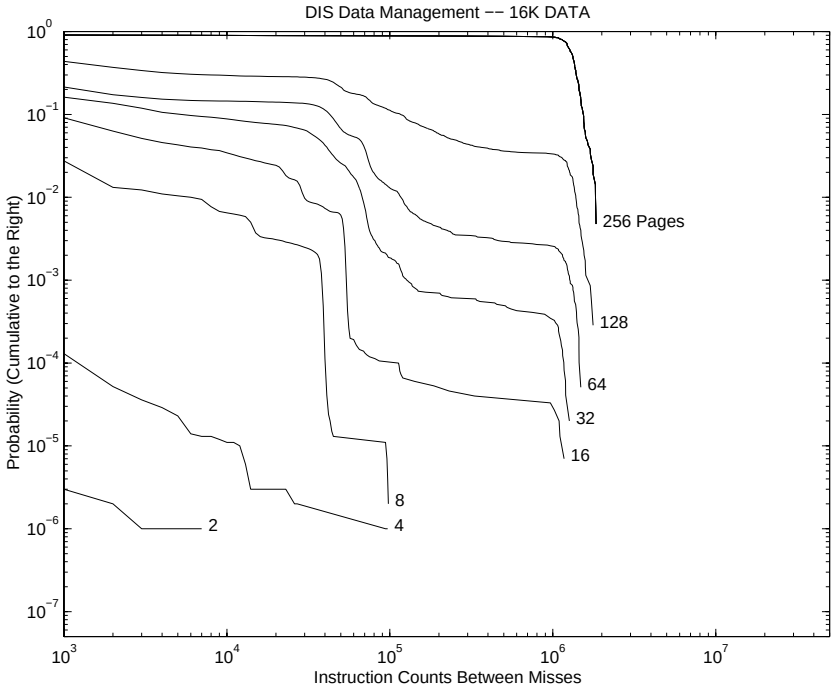


Figure B.3. DIS Data Management 16 KB Page CIPD (Ψ)

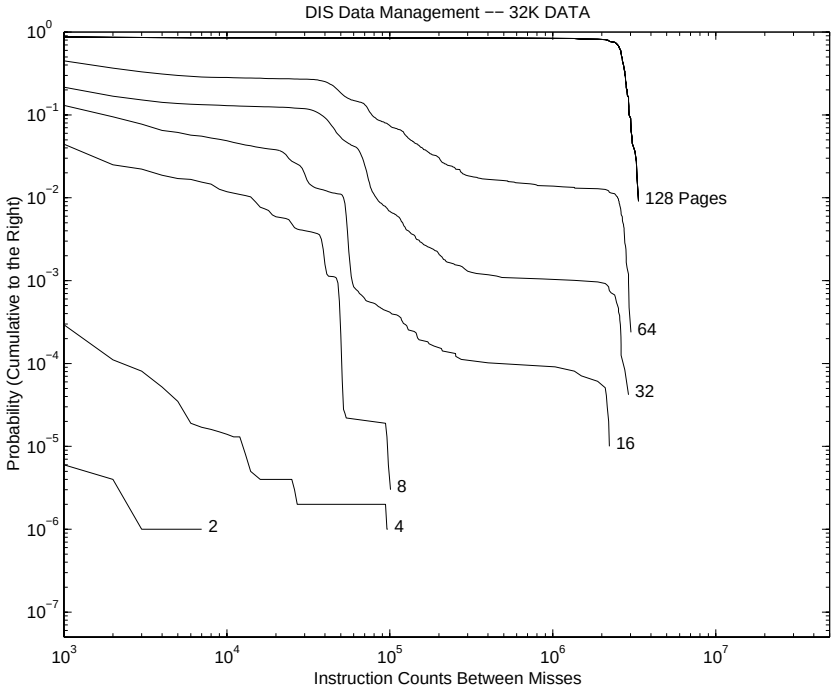


Figure B.4. DIS Data Management 32 KB Page CIPD (Ψ)

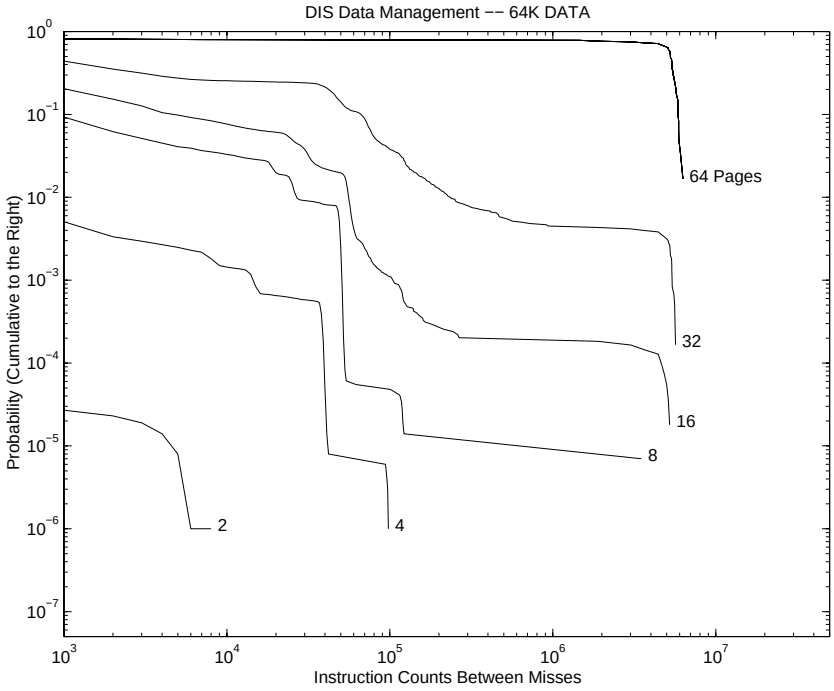


Figure B.5. DIS Data Management 64 KB Page CIPD (Ψ)

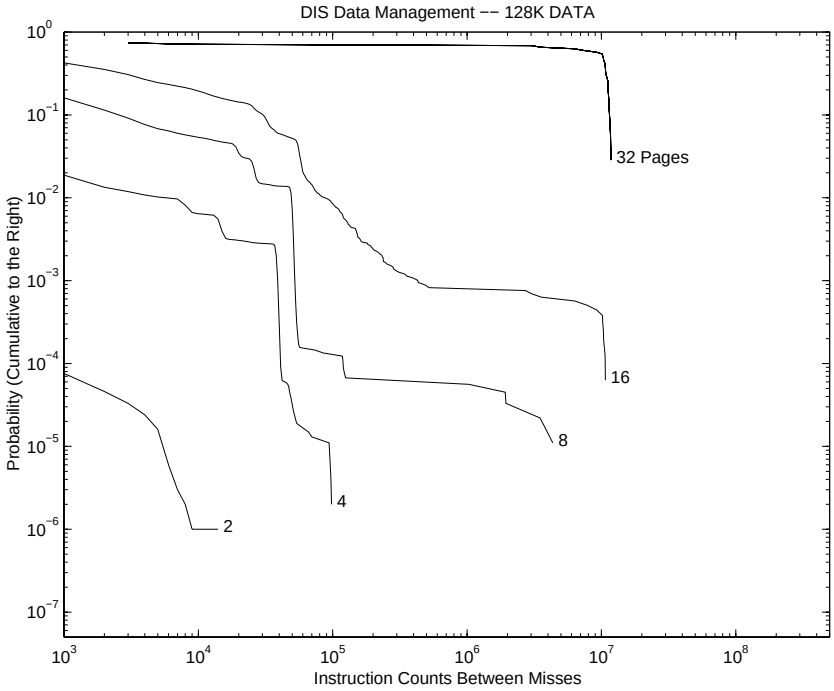


Figure B.6. DIS Data Management 128 KB Page CIPD (Ψ)

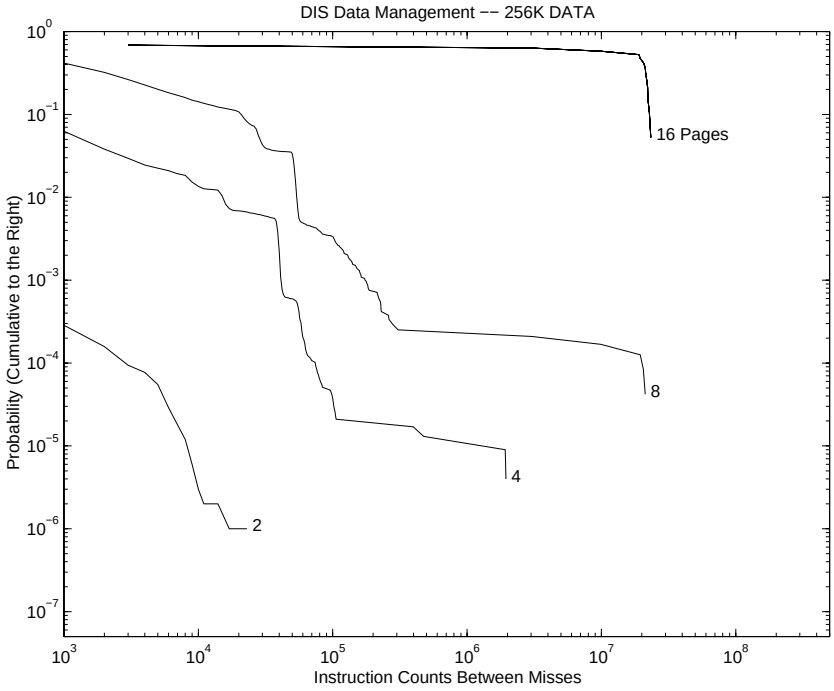


Figure B.7. DIS Data Management 256 KB Page CIPD (Ψ)

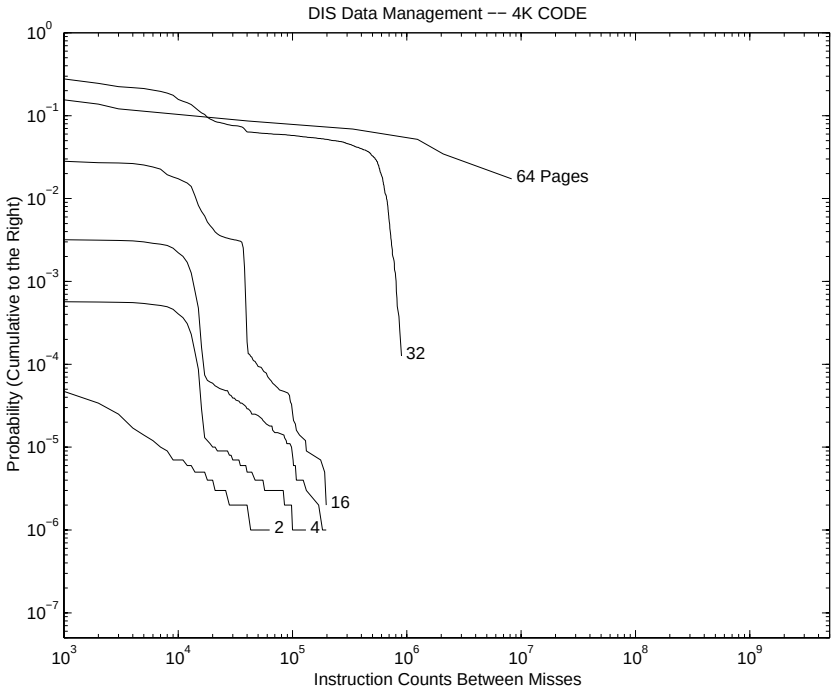


Figure B.8. DIS Data Management 4 KB (Code and Stack) KB Page CIPD (Ψ)

B.1.2 Cache Configurations

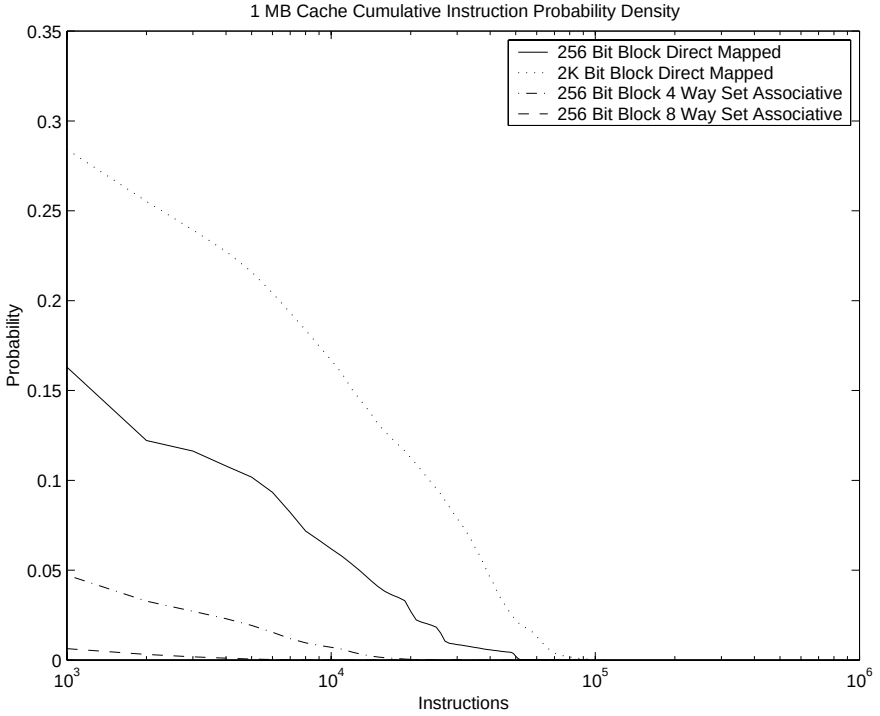


Figure B.9. DIS Data Management 1 MB Data Cache CIPD (Ψ)

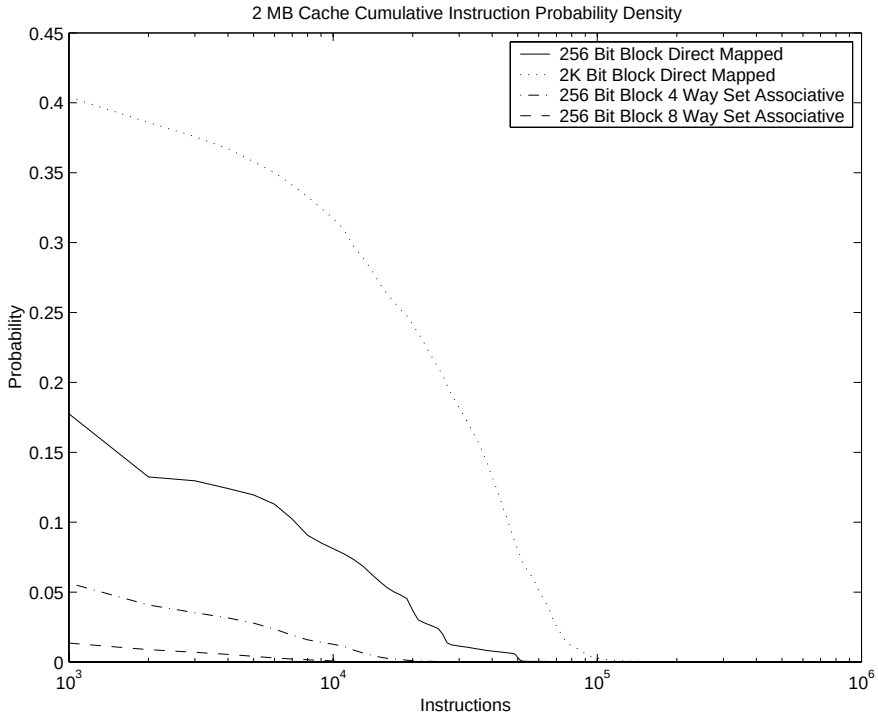


Figure B.10. DIS Data Management 2 MB Data Cache CIPD (Ψ)

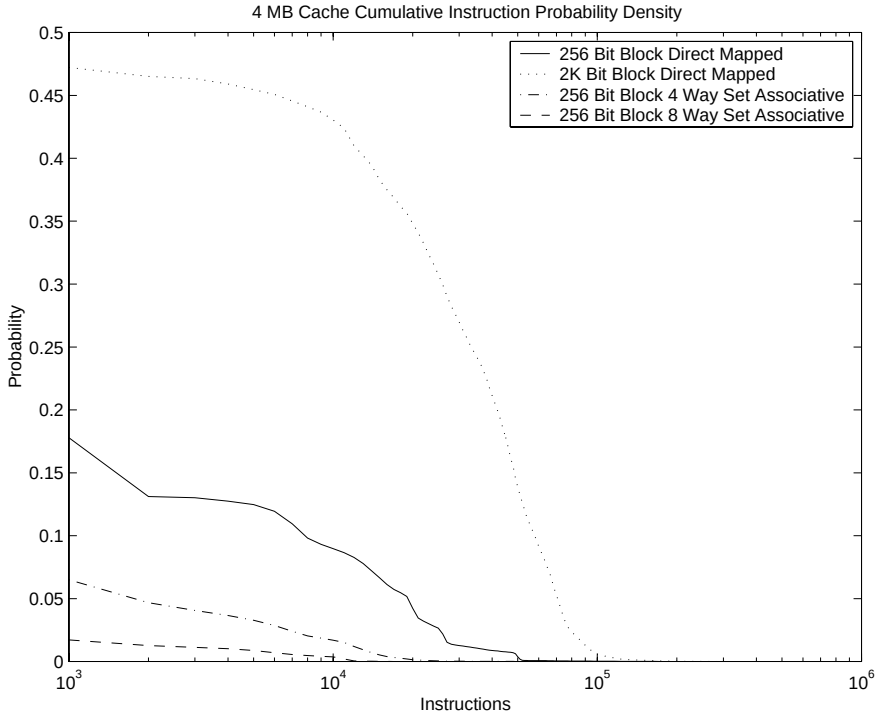


Figure B.11. DIS Data Management 4 MB Data Cache CIPD (Ψ)

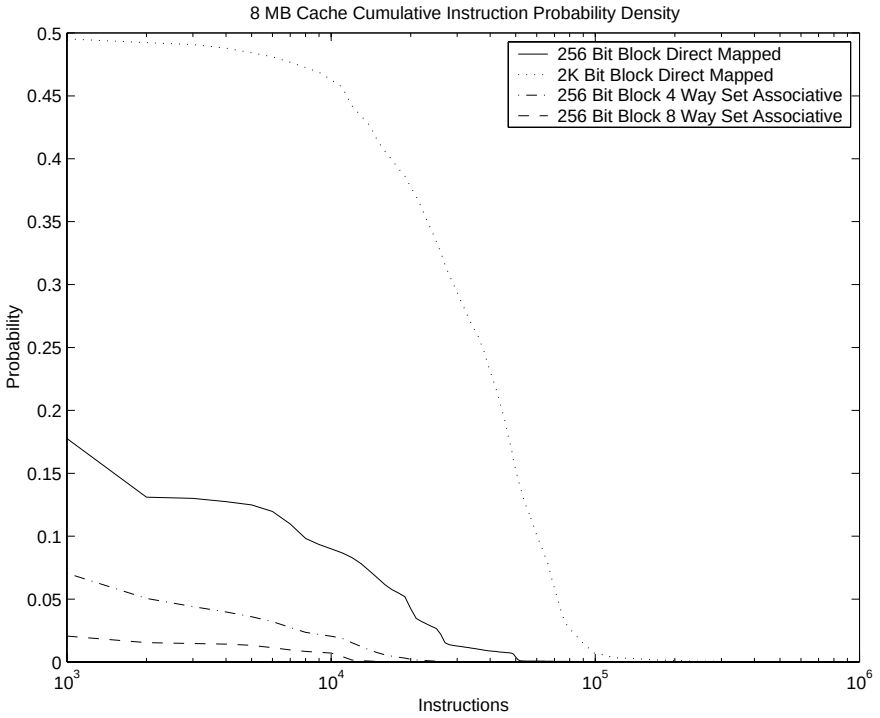


Figure B.12. DIS Data Management 8 MB Data Cache CIPD (Ψ)

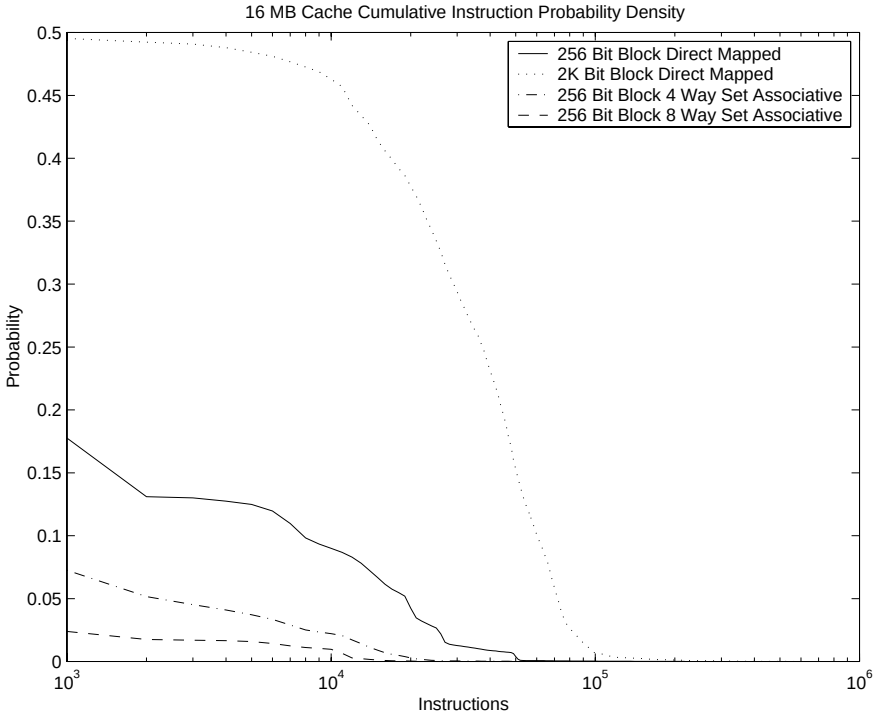


Figure B.13. DIS Data Management 16 MB Data Cache CIPD (Ψ)

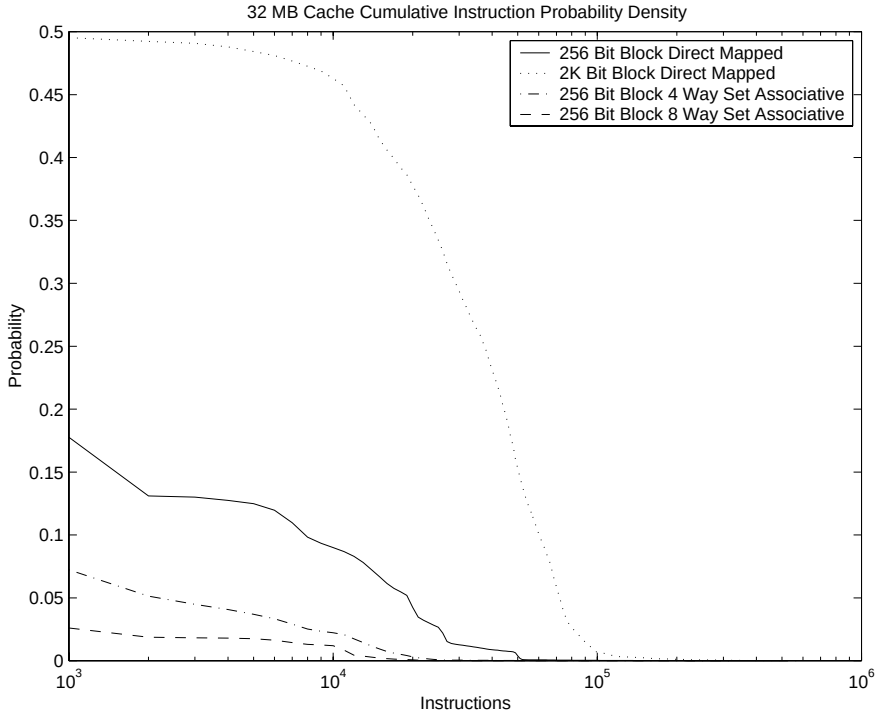


Figure B.14. DIS Data Management 32 MB Data Cache CIPD (Ψ)

B.2 DIS FFT

B.2.1 Page Configurations

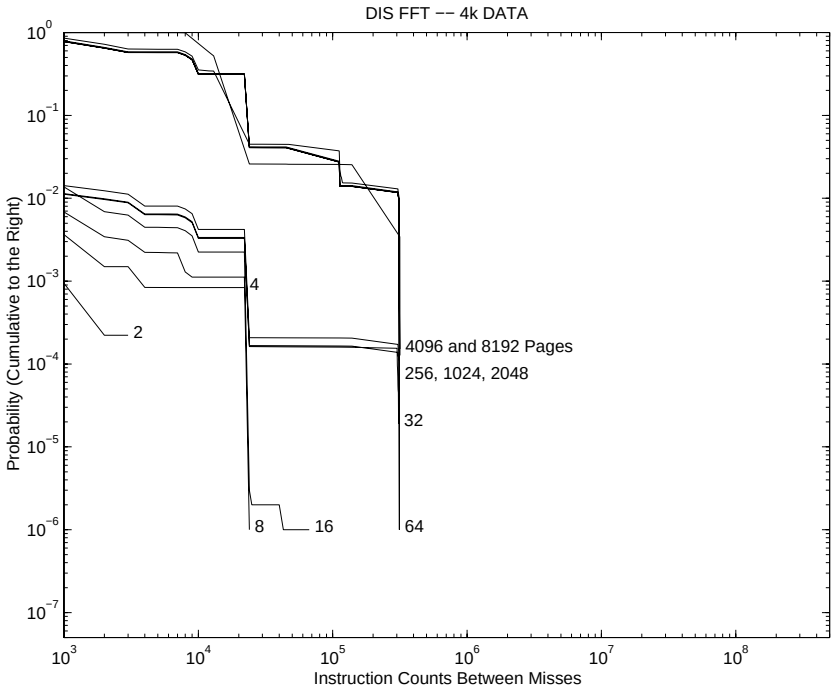


Figure B.15. DIS FFT 4 KB Page CIPD (Ψ)

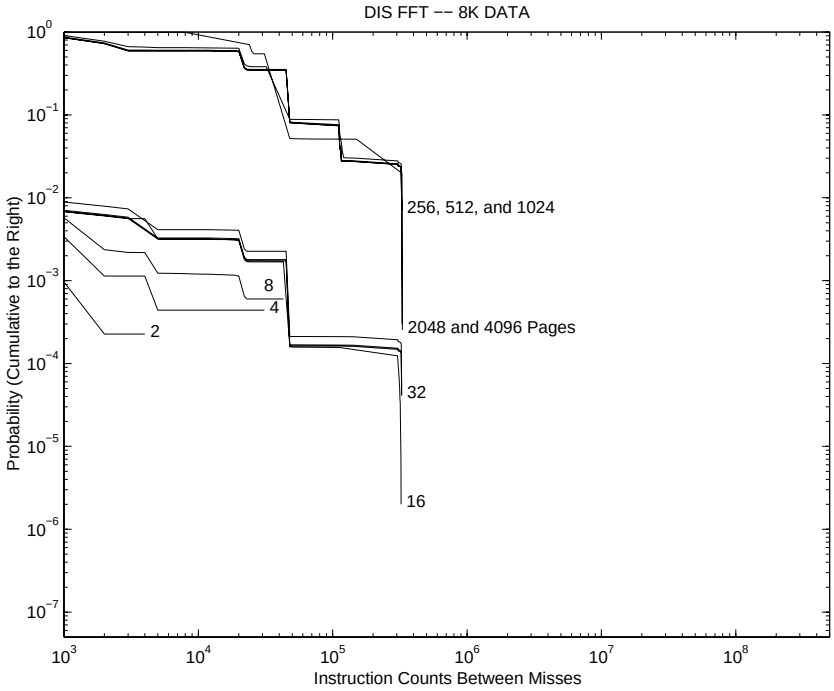


Figure B.16. DIS FFT 8 KB Page CIPD (Ψ)

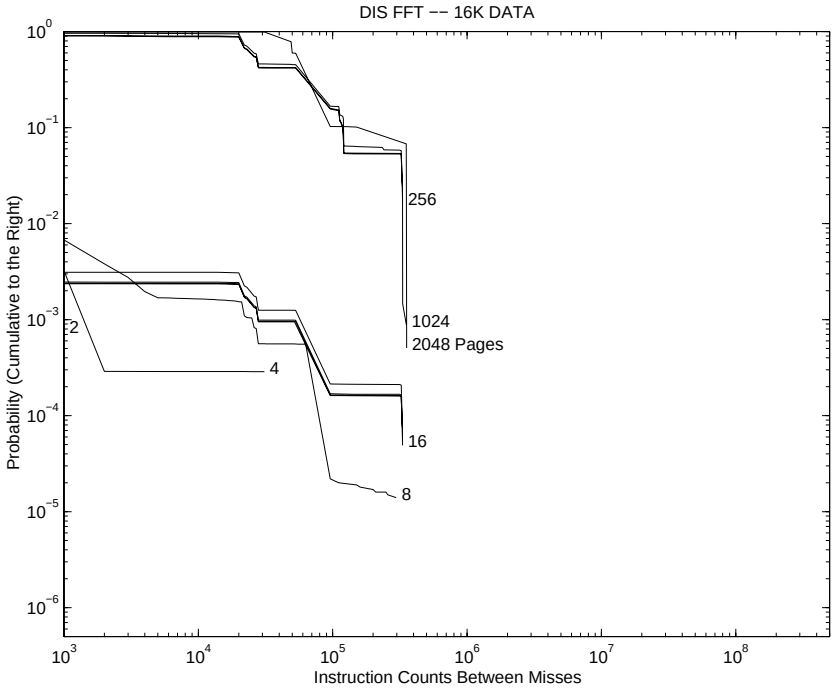


Figure B.17. DIS FFT 16 KB Page CIPD (Ψ)

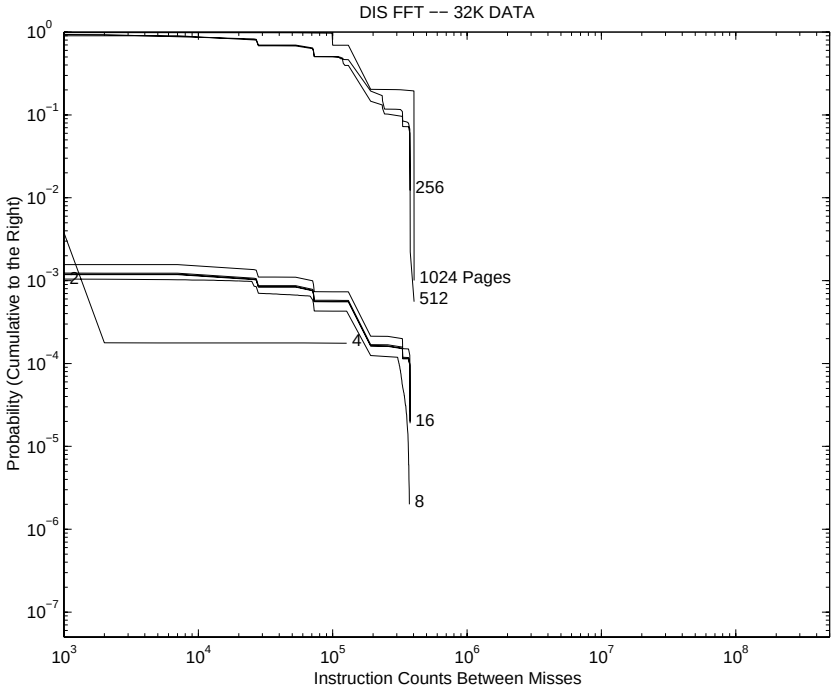


Figure B.18. DIS FFT 32 KB Page CIPD (Ψ)

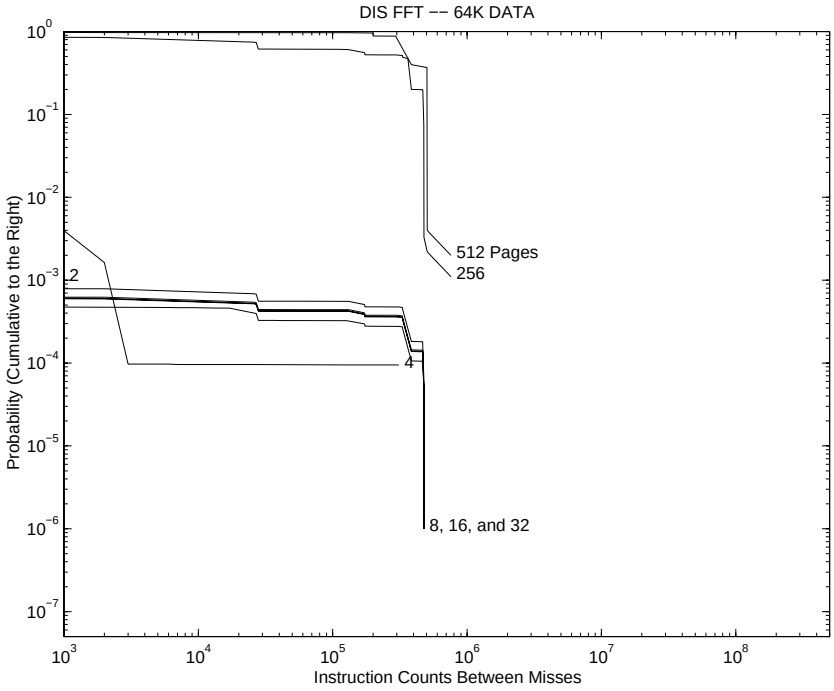


Figure B.19. DIS FFT 64 KB Page CIPD (Ψ)

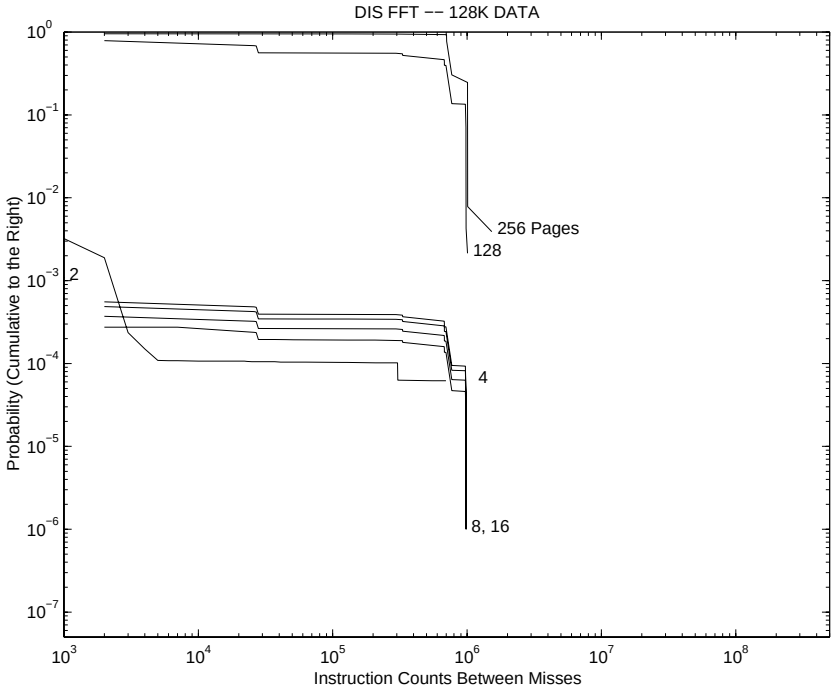


Figure B.20. DIS FFT 128 KB Page CIPD (Ψ)

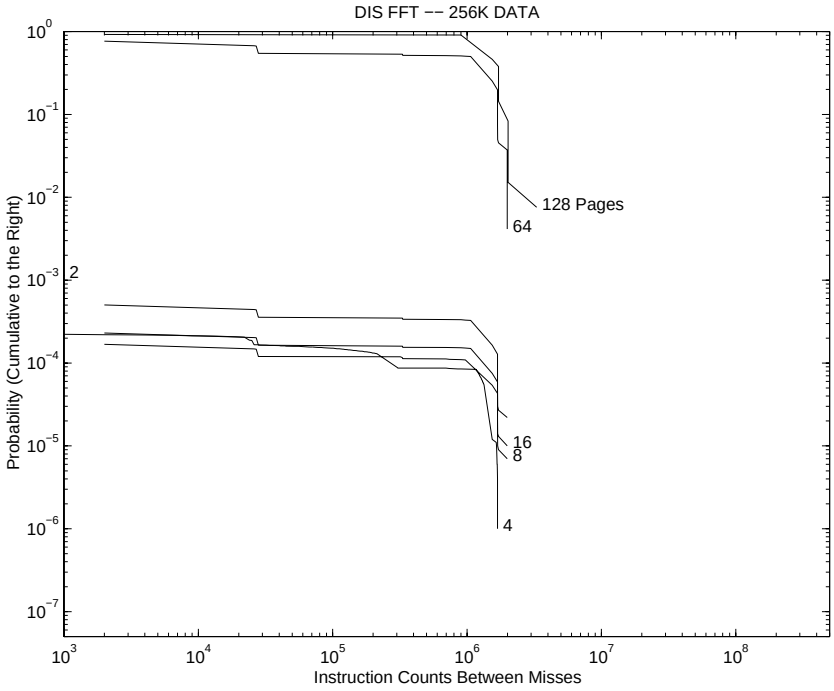


Figure B.21. DIS FFT 256 KB Page CIPD (Ψ)

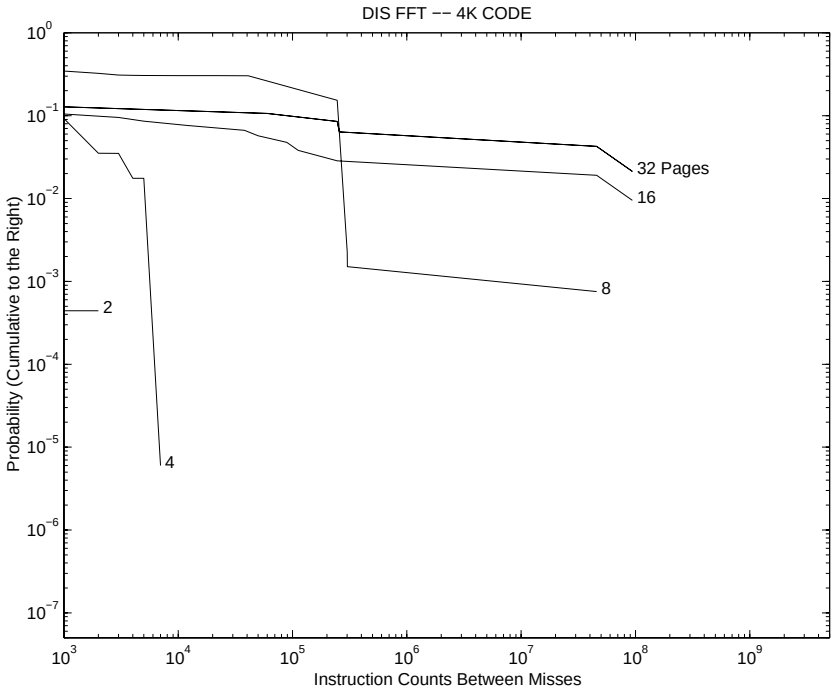


Figure B.22. DIS FFT 4 KB (Code and Stack) KB Page CIPD (Ψ)

B.2.2 Cache Configurations

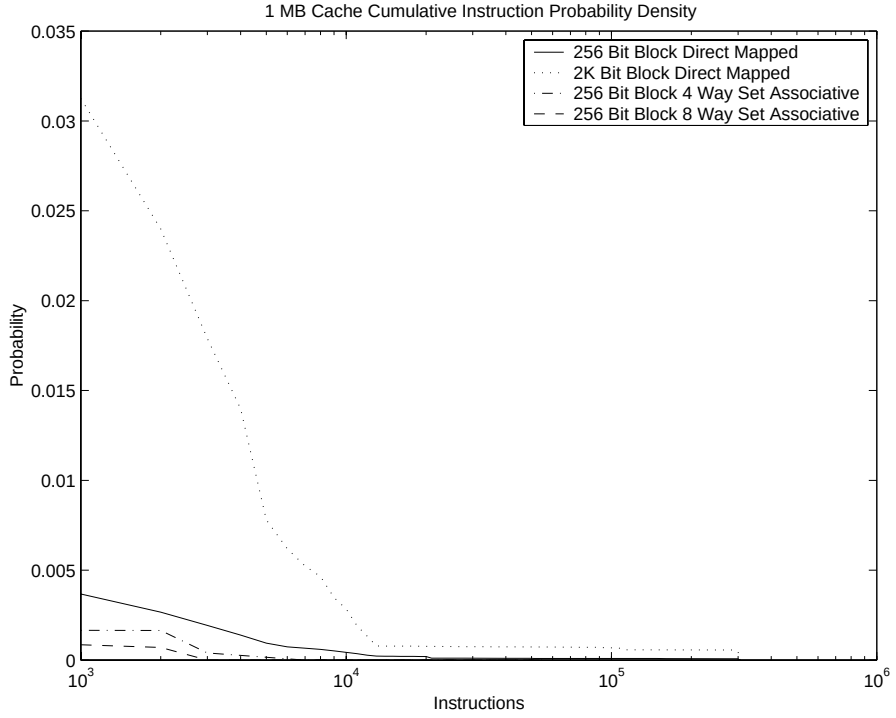


Figure B.23. DIS FFT 1 MB Data Cache CIPD (Ψ)

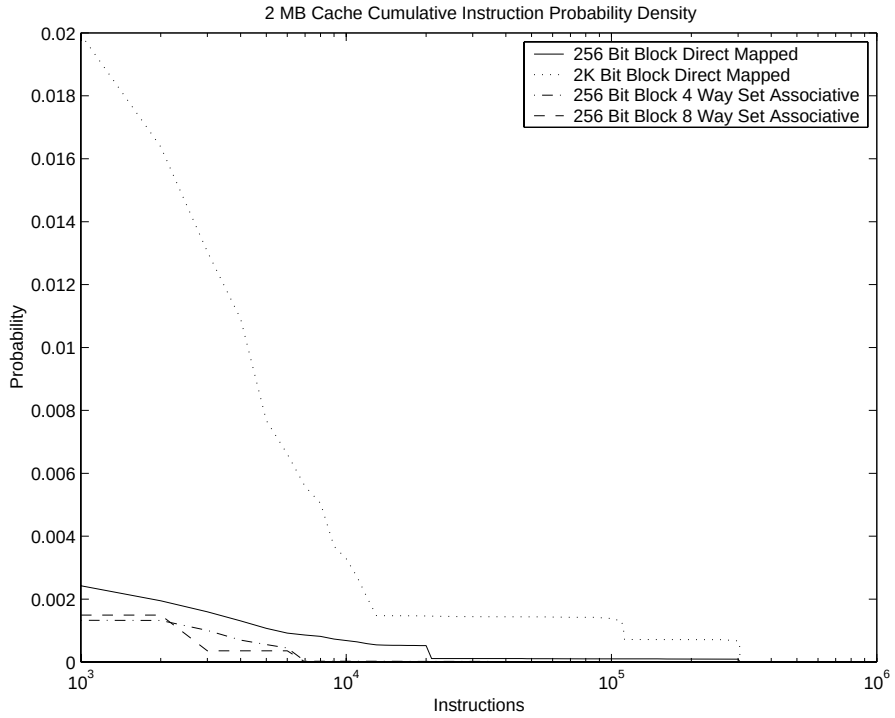


Figure B.24. DIS FFT 2 MB Data Cache CIPD (Ψ)

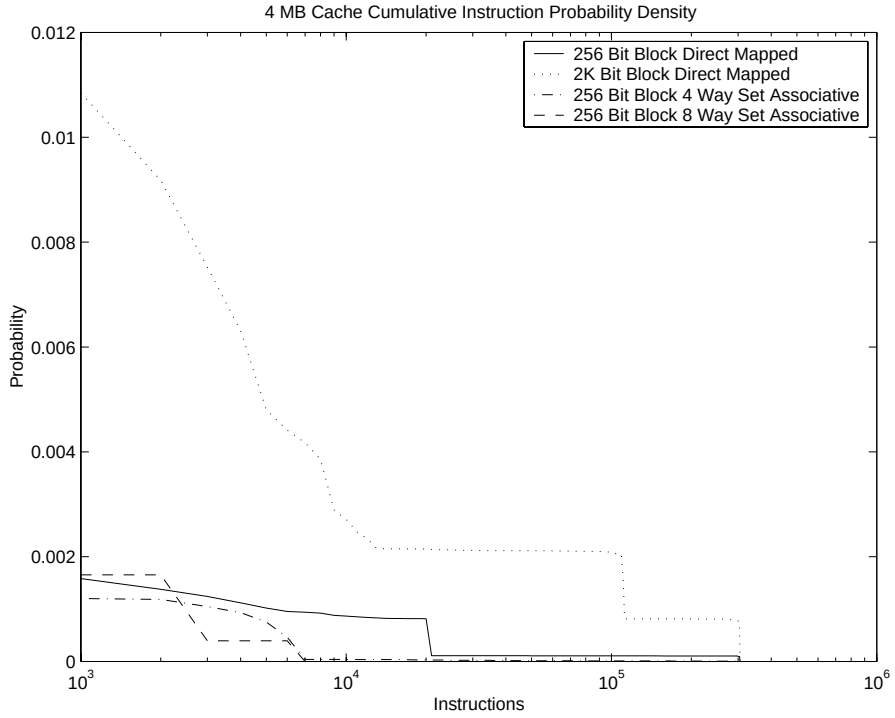


Figure B.25. DIS FFT 4 MB Data Cache CIPD (Ψ)

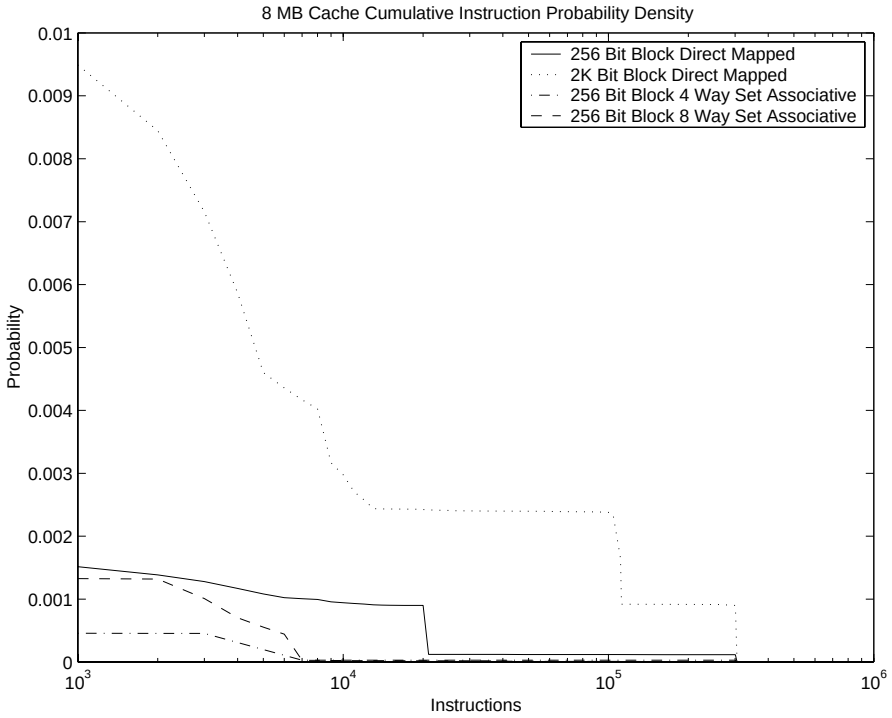


Figure B.26. DIS FFT 8 MB Data Cache CIPD (Ψ)

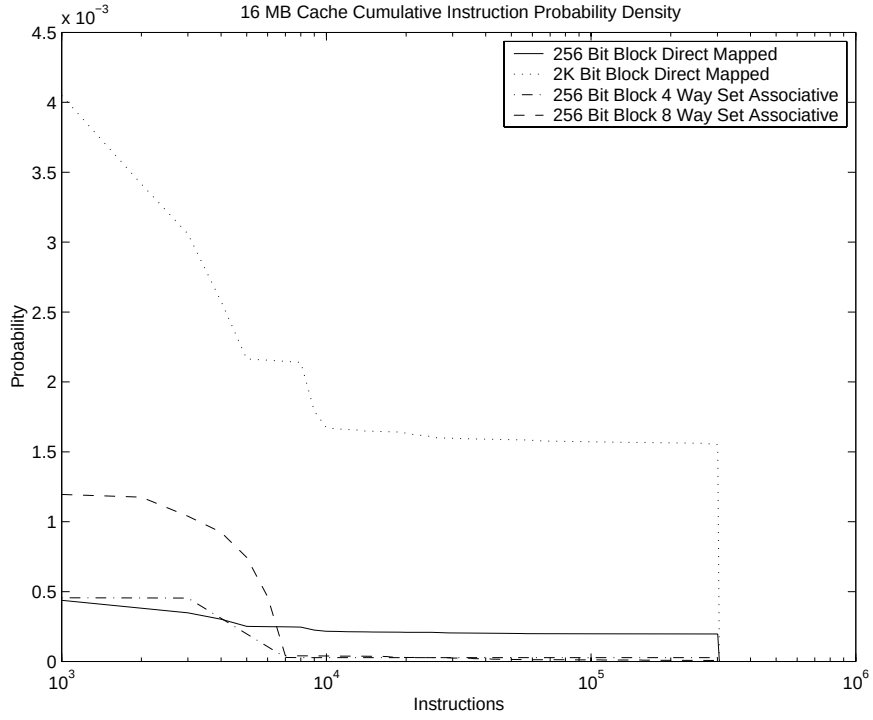


Figure B.27. DIS FFT 16 MB Data Cache CIPD (Ψ)

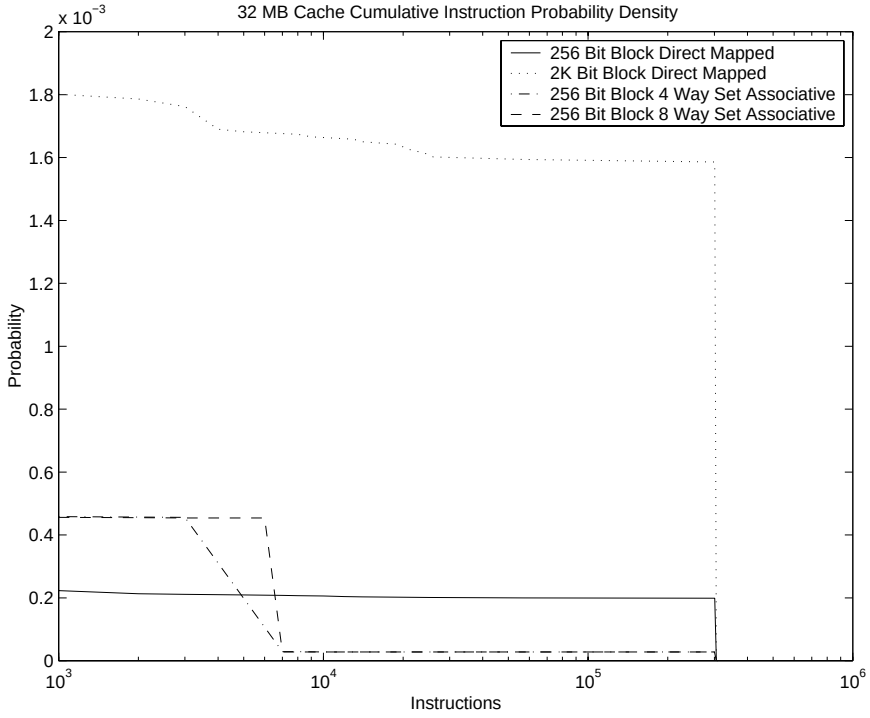


Figure B.28. DIS FFT 32 MB Data Cache CIPD (Ψ)

B.3 DIS Method of Moments

B.3.1 Page Configurations

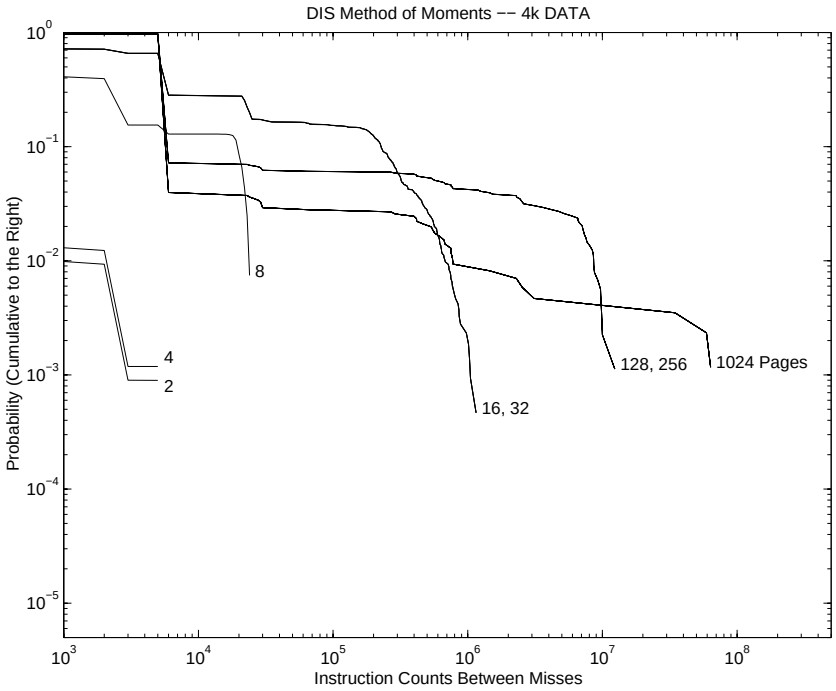


Figure B.29. DIS Method of Moments 4 KB Page CIPD (Ψ)

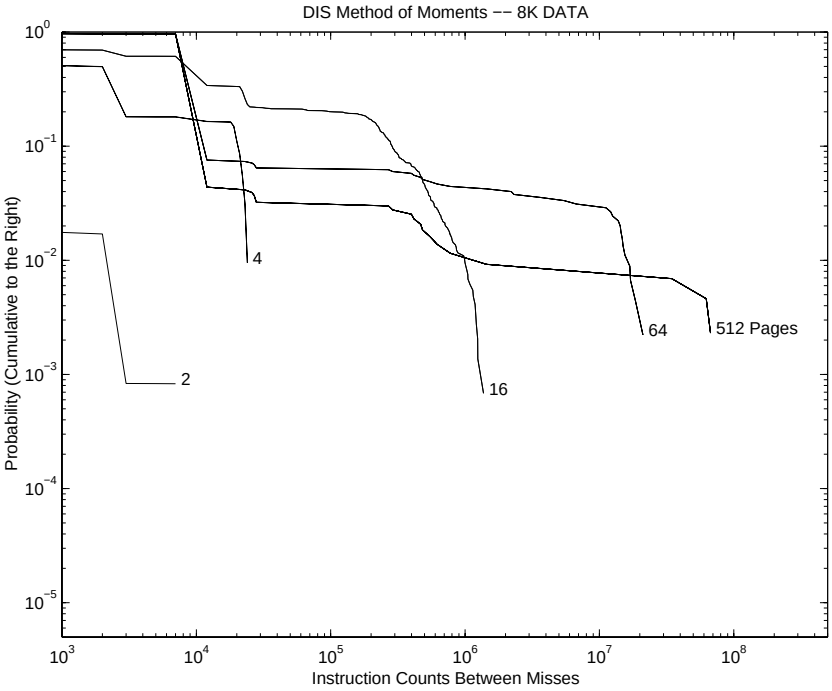


Figure B.30. DIS Method of Moments 8 KB Page CIPD (Ψ)

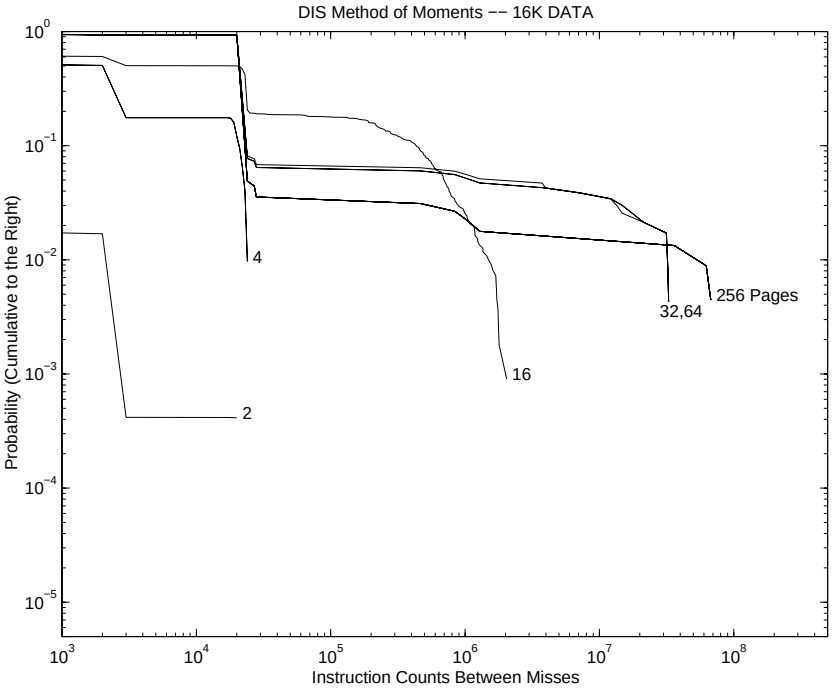


Figure B.31. DIS Method of Moments 16 KB Page CIPD (Ψ)

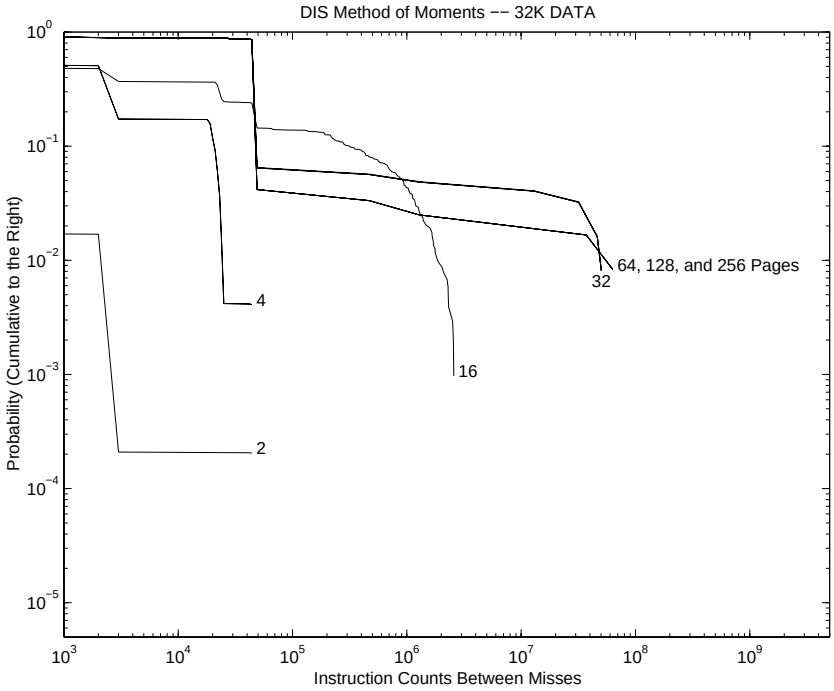


Figure B.32. DIS Method of Moments 32 KB Page CIPD (Ψ)

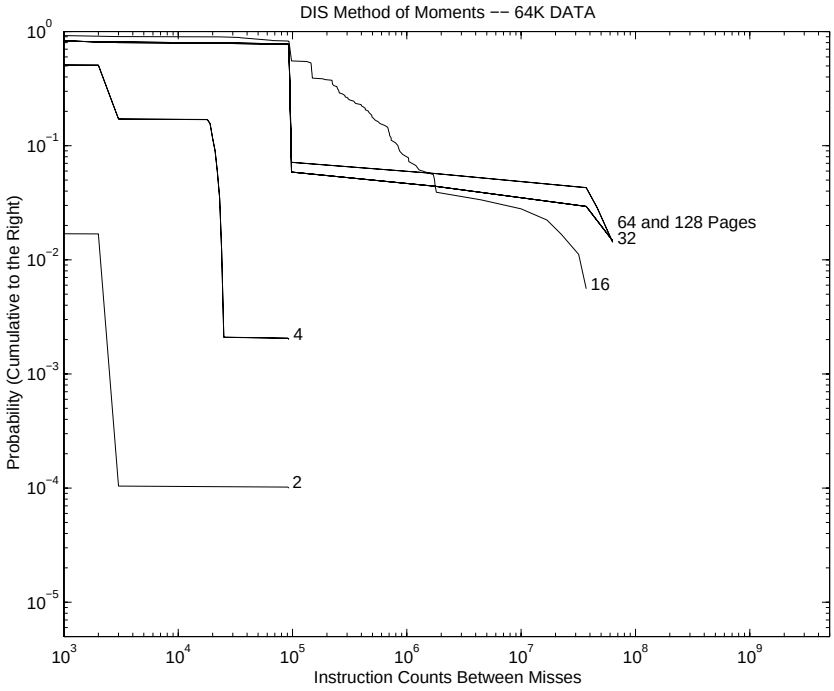


Figure B.33. DIS Method of Moments 64 KB Page CIPD (Ψ)

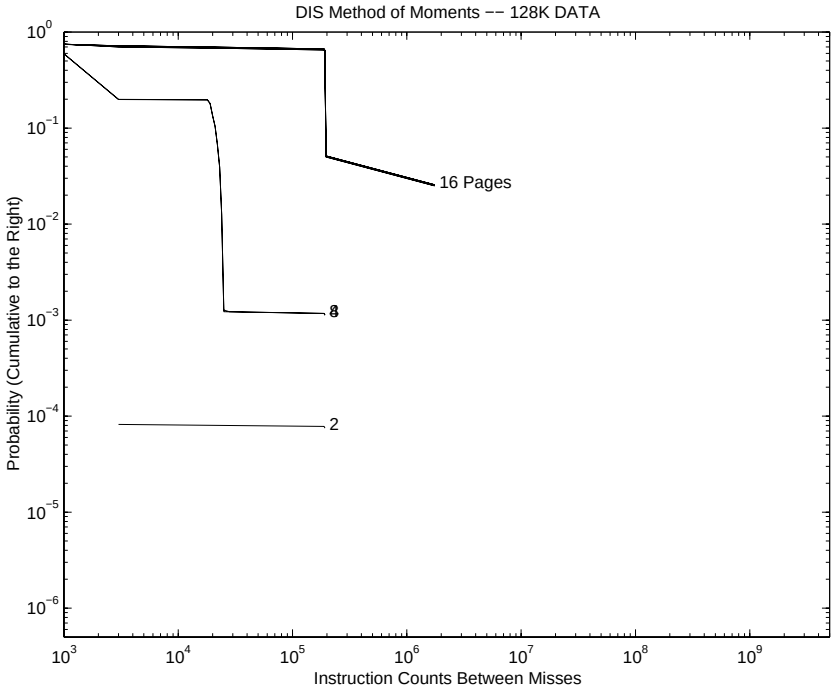


Figure B.34. DIS Method of Moments 128 KB Page CIPD (Ψ)

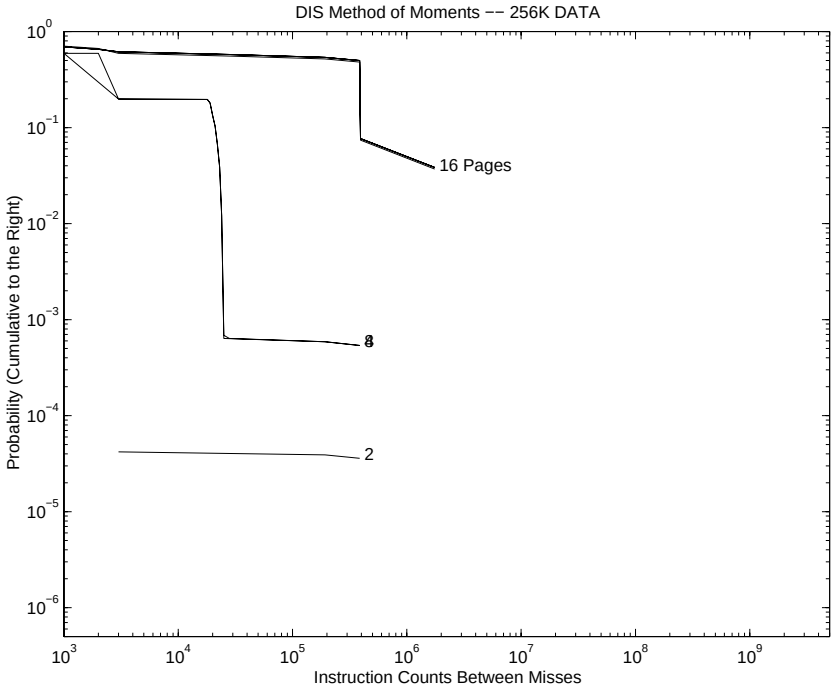


Figure B.35. DIS Method of Moments 256 KB Page CIPD (Ψ)

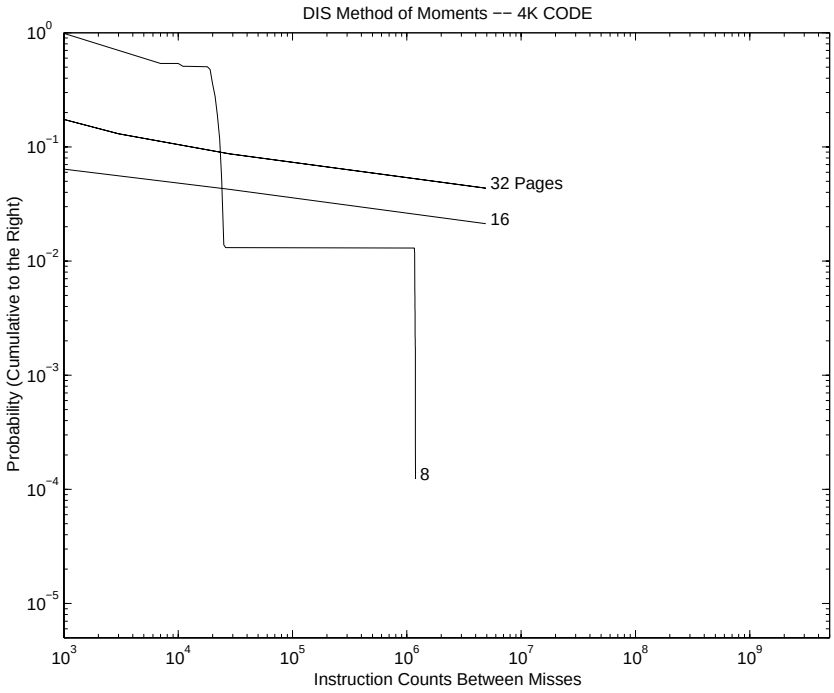


Figure B.36. DIS Method of Moments 4 KB (Code and Stack) KB Page CIPD (Ψ)

B.3.2 Cache Configurations

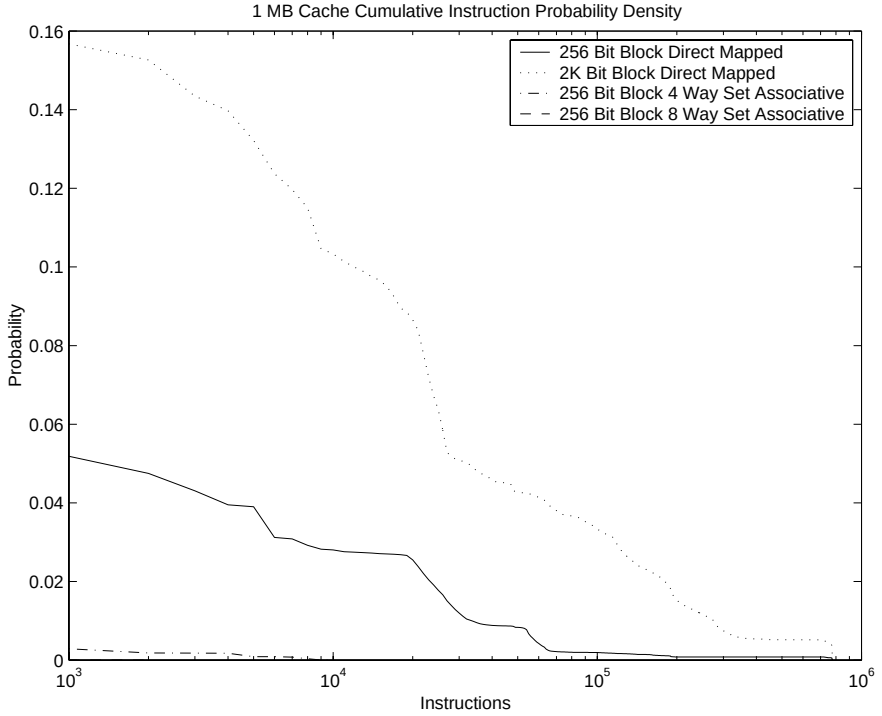


Figure B.37. DIS Method of Moments 1 MB Data Cache CIPD (Ψ)

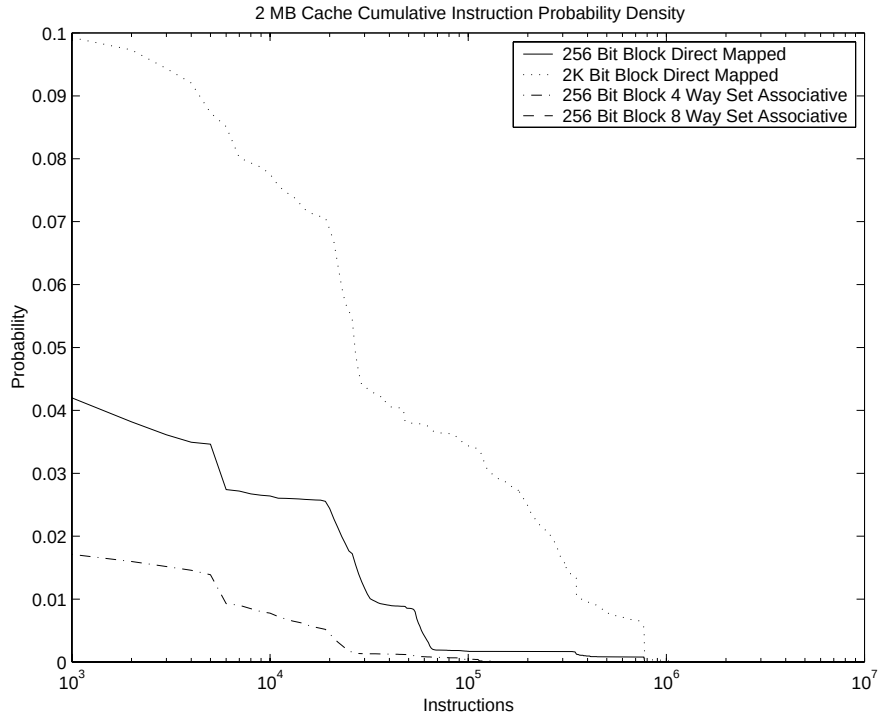


Figure B.38. DIS Method of Moments 2 MB Data Cache CIPD (Ψ)

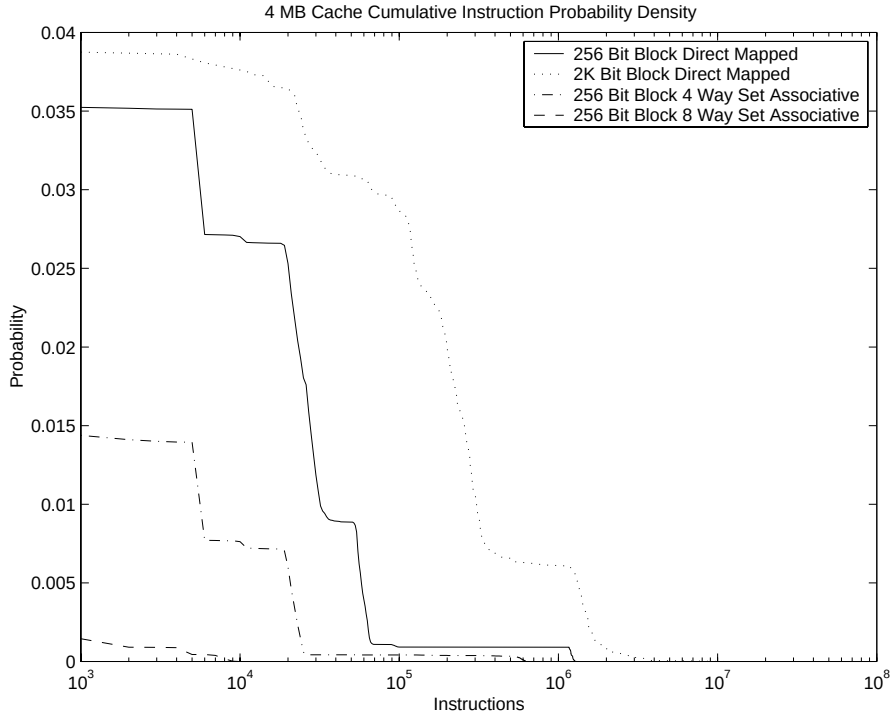


Figure B.39. DIS Method of Moments 4 MB Data Cache CIPD (Ψ)

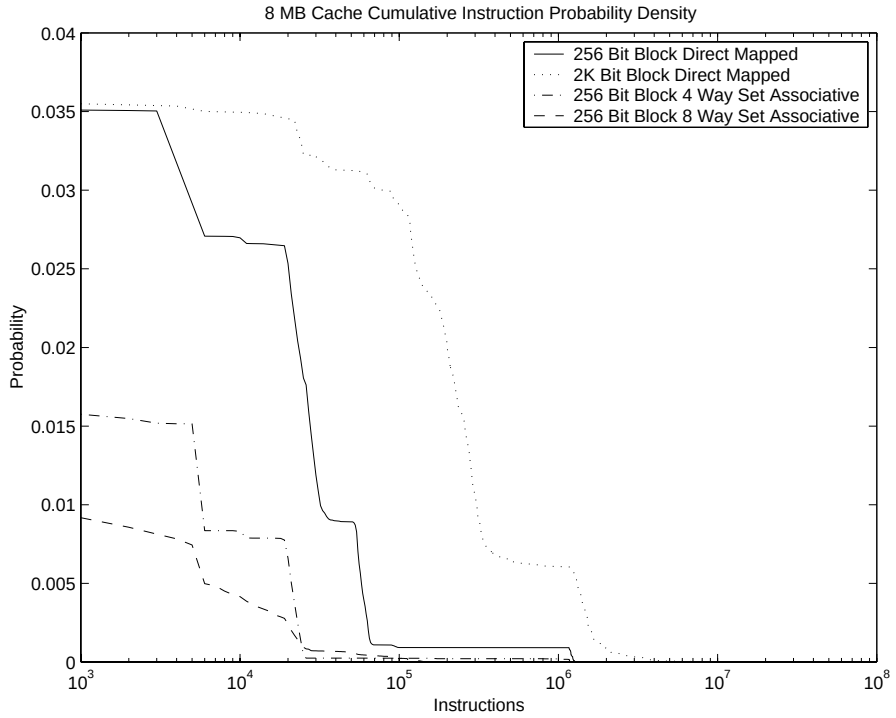


Figure B.40. DIS Method of Moments 8 MB Data Cache CIPD (Ψ)

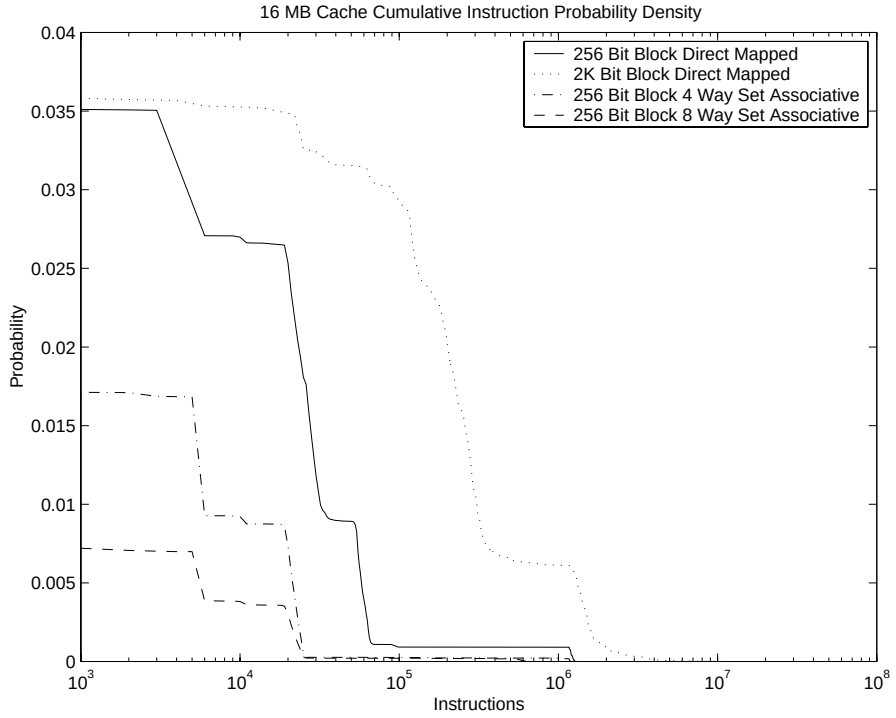


Figure B.41. DIS Method of Moments 16 MB Data Cache CIPD (Ψ)

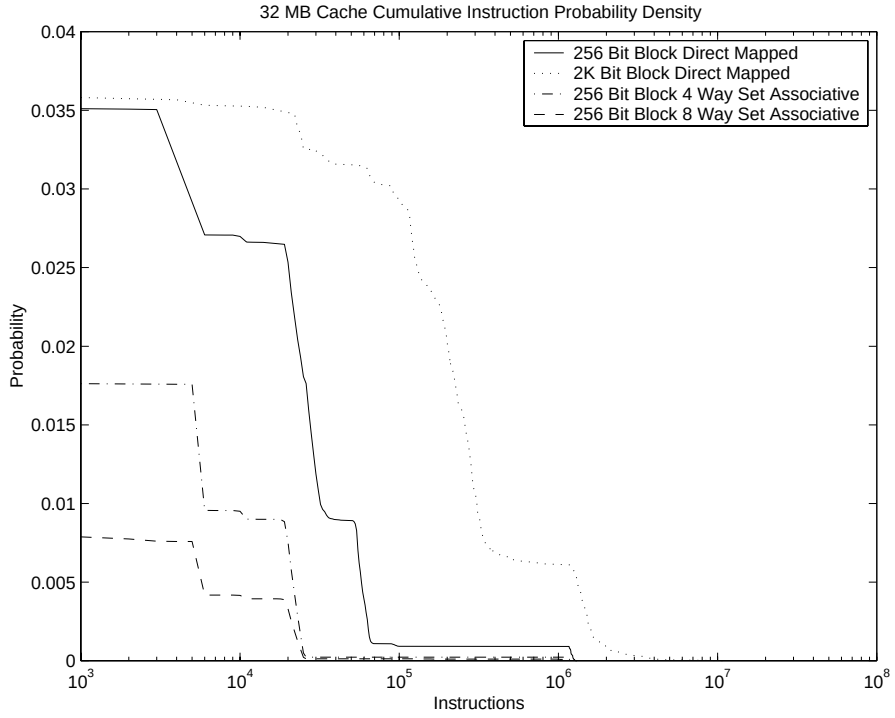


Figure B.42. DIS Method of Moments 32 MB Data Cache CIPD (Ψ)

B.4 DIS Image Understanding

B.4.1 Page Configurations

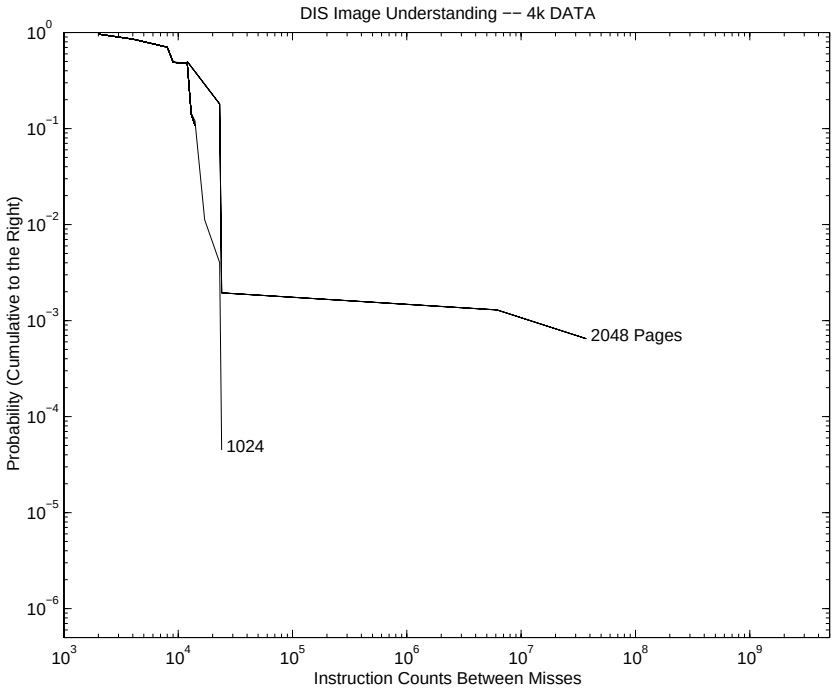


Figure B.43. DIS Image Understanding 4 KB Page CIPD (Ψ)

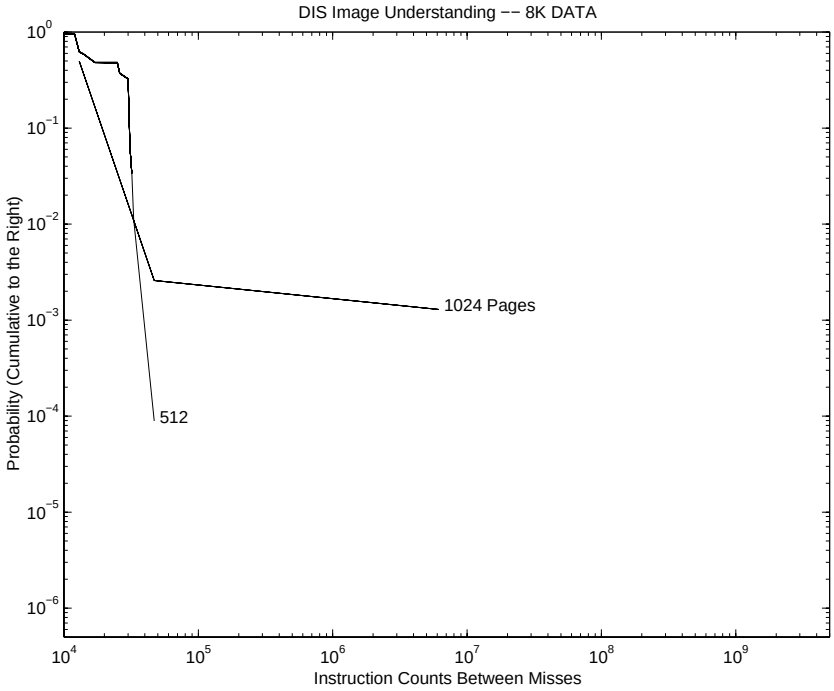


Figure B.44. DIS Image Understanding 8 KB Page CIPD (Ψ)

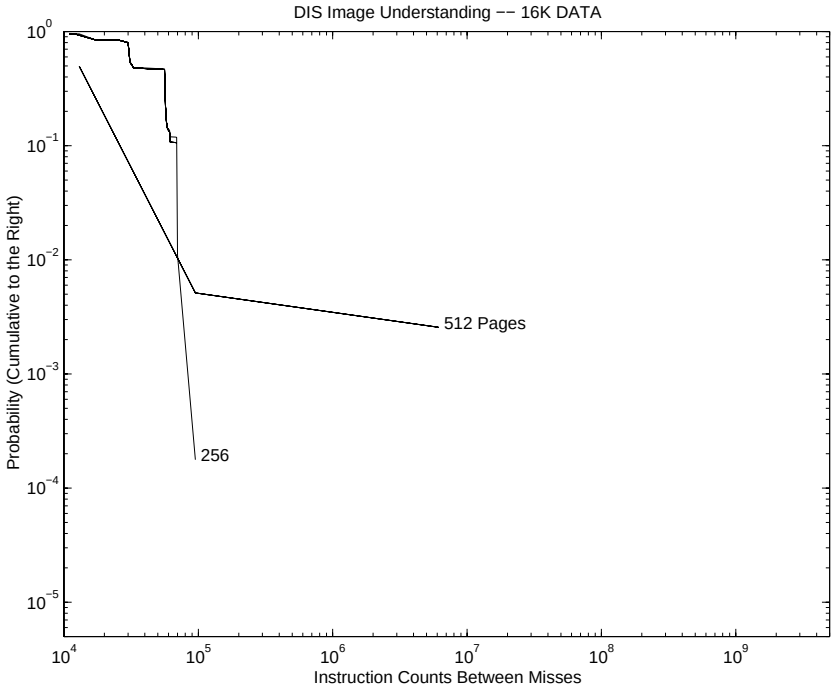


Figure B.45. DIS Image Understanding 16 KB Page CIPD (Ψ)

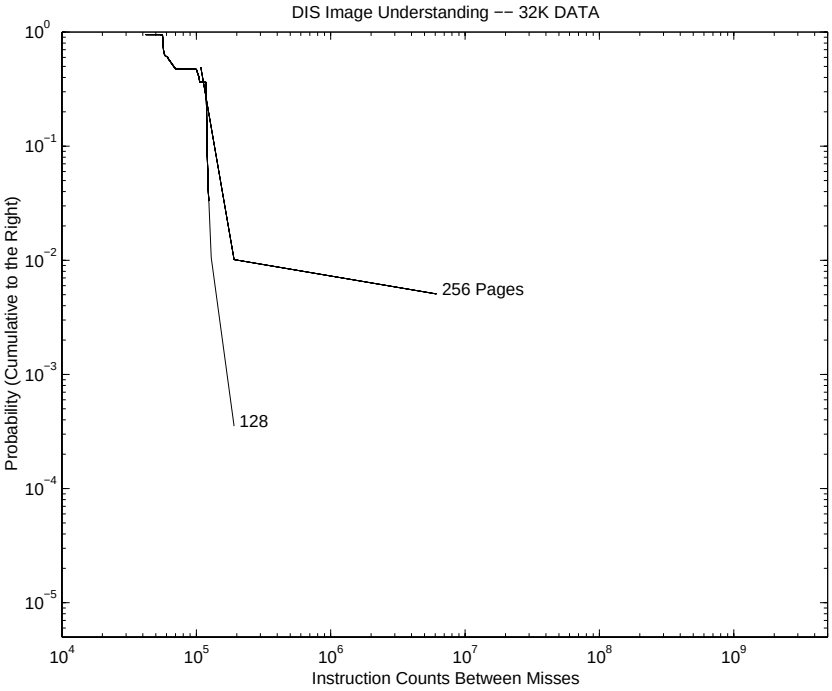


Figure B.46. DIS Image Understanding 32 KB Page CIPD (Ψ)

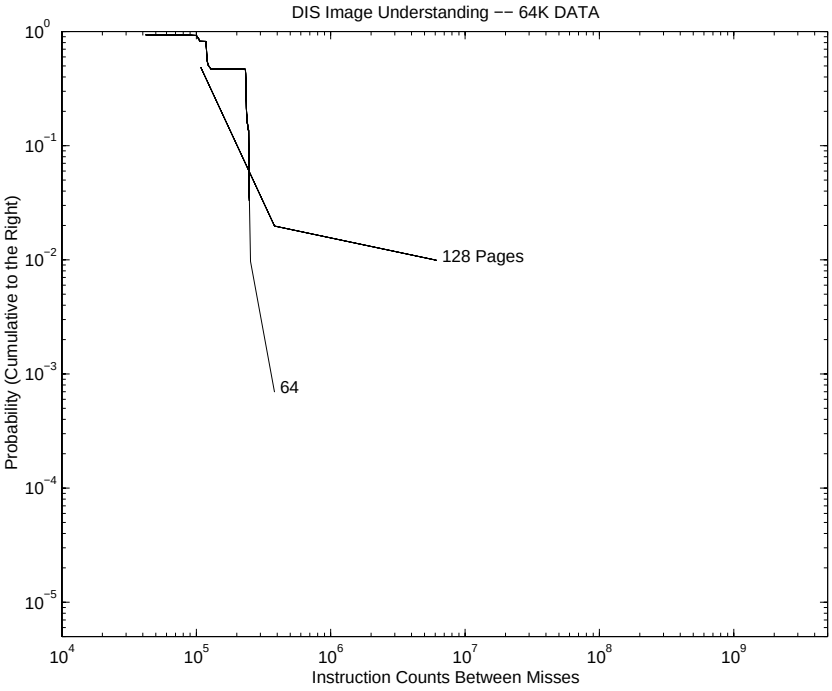


Figure B.47. DIS Image Understanding 64 KB Page CIPD (Ψ)

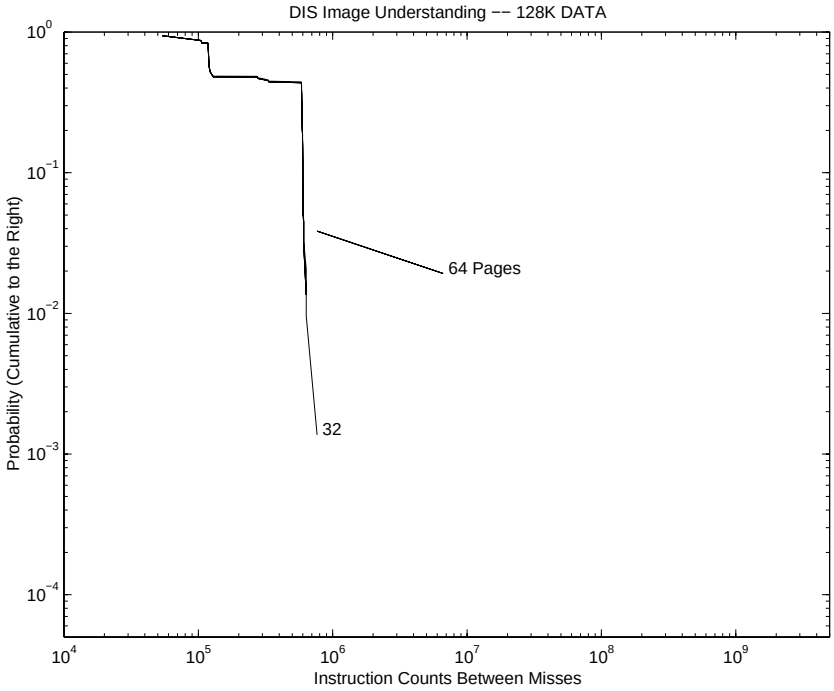


Figure B.48. DIS Image Understanding 128 KB Page CIPD (Ψ)

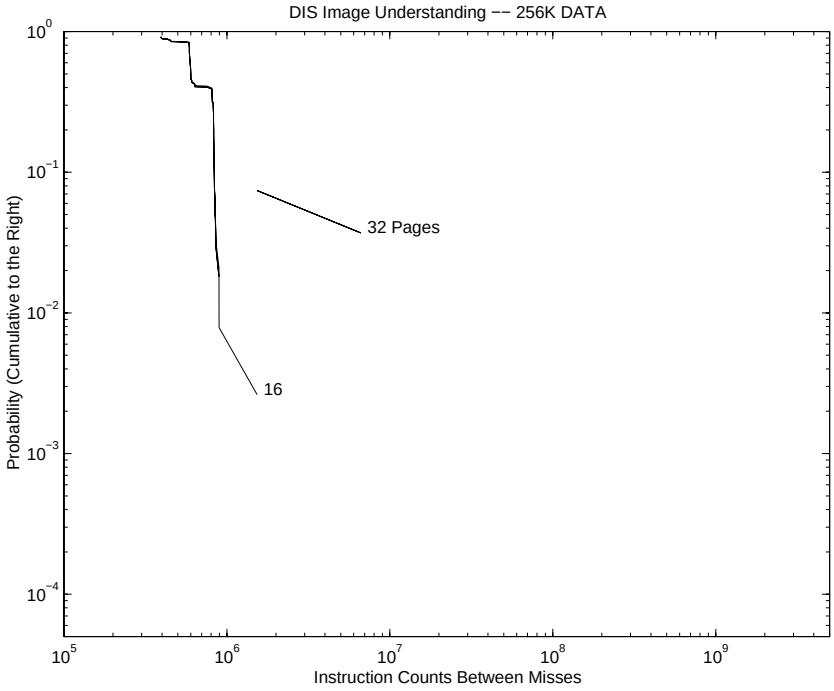


Figure B.49. DIS Image Understanding 256 KB Page CIPD (Ψ)

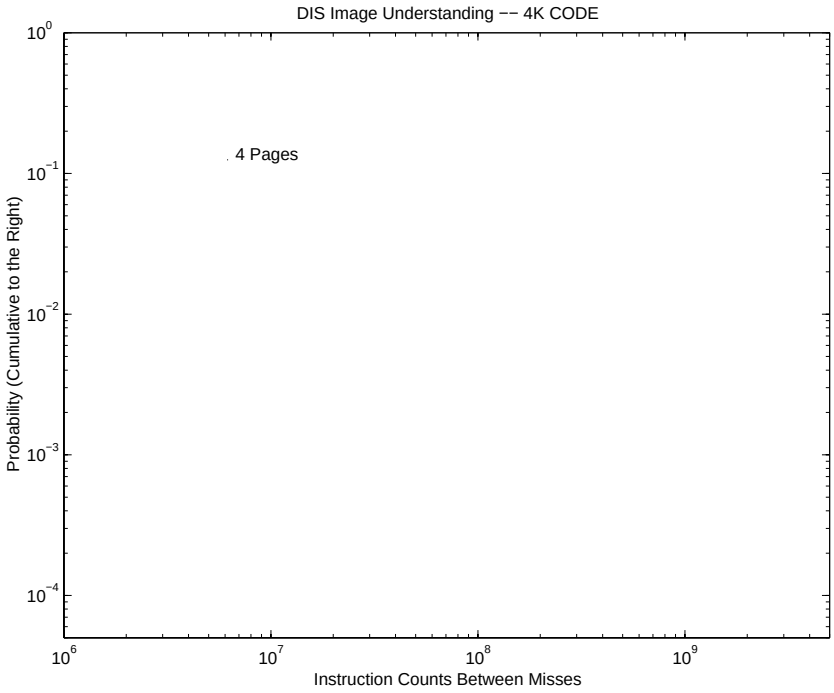


Figure B.50. DIS Image Understanding 4 KB (Code and Stack) KB Page CIPD (Ψ)

B.4.2 Cache Configurations

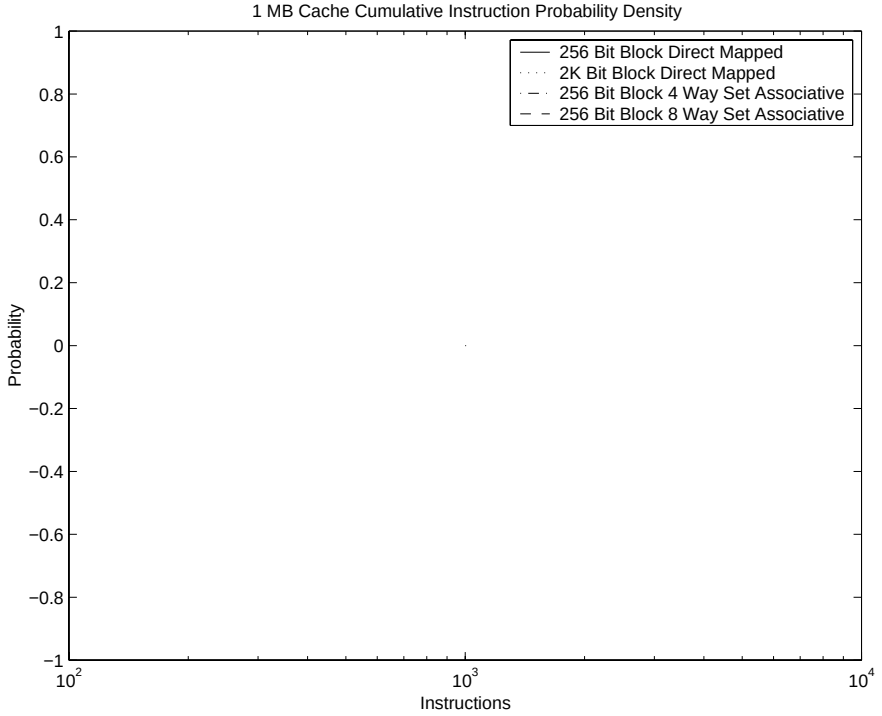


Figure B.51. DIS Image Understanding 1 MB Data Cache CIPD (Ψ)

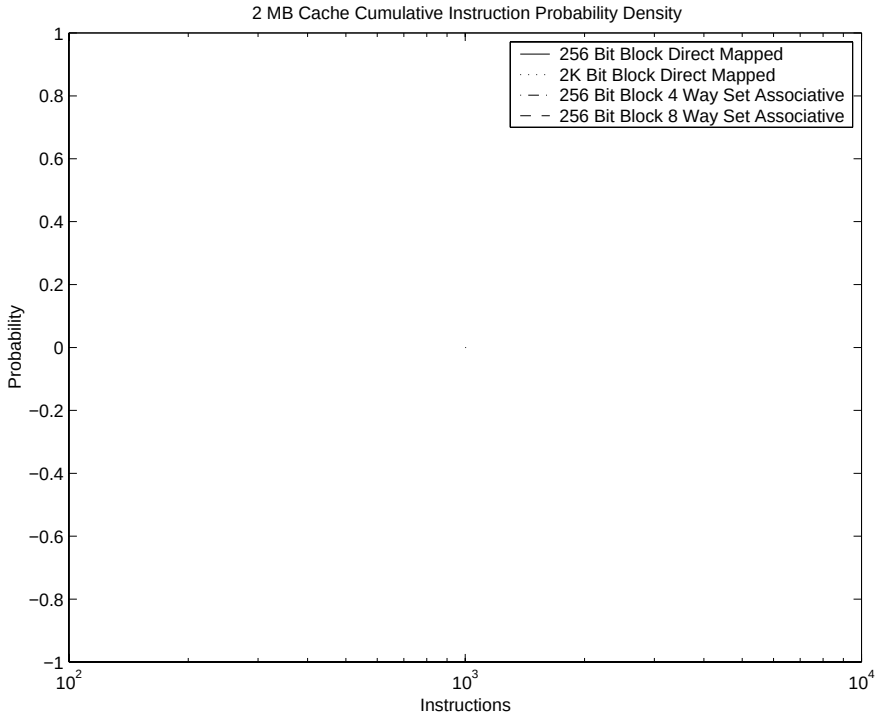


Figure B.52. DIS Image Understanding 2 MB Data Cache CIPD (Ψ)

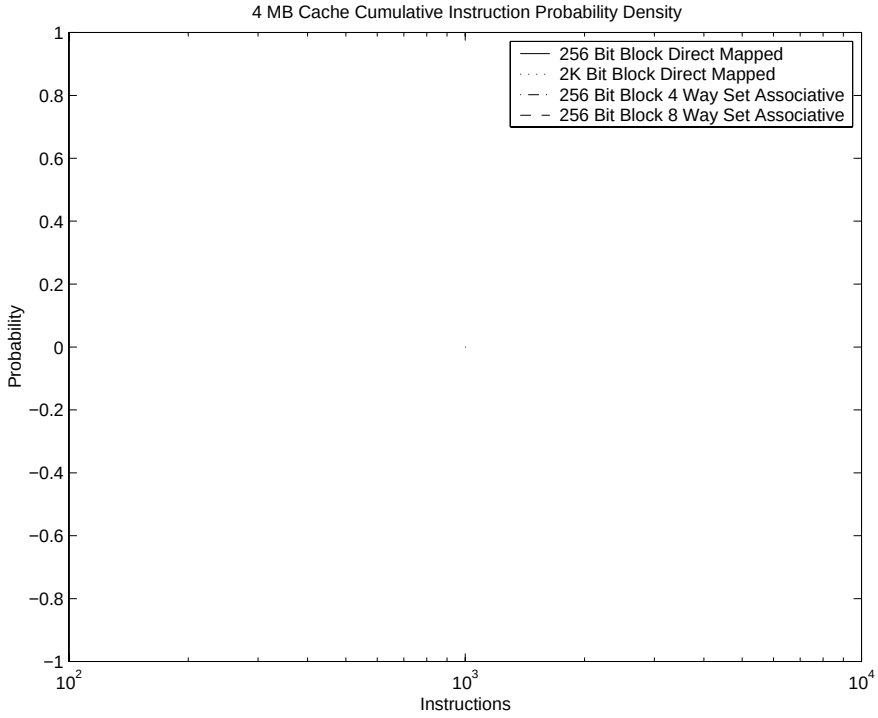


Figure B.53. DIS Image Understanding 4 MB Data Cache CIPD (Ψ)

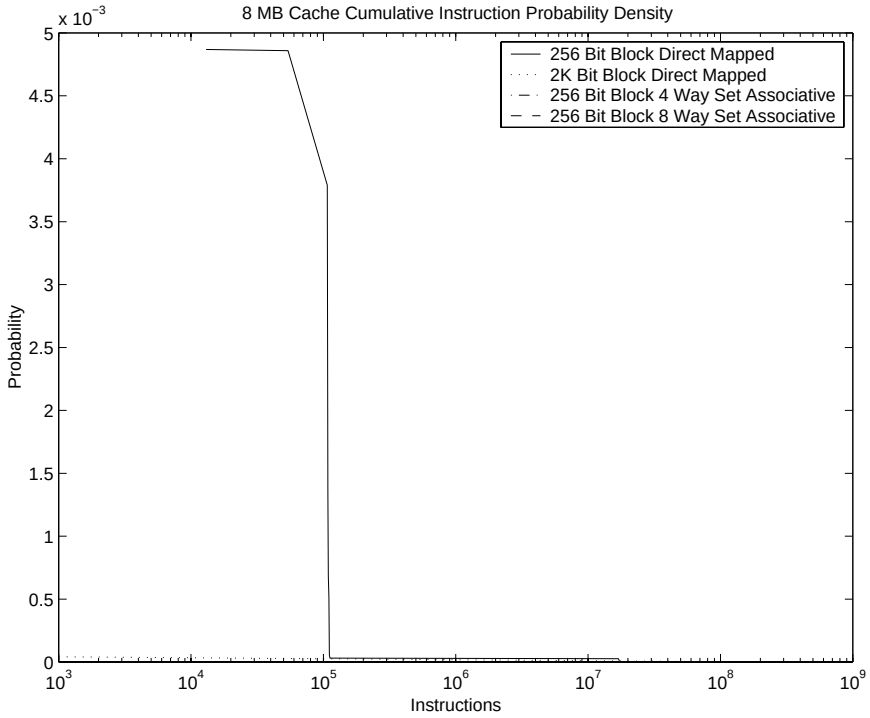


Figure B.54. DIS Image Understanding 8 MB Data Cache CIPD (Ψ)

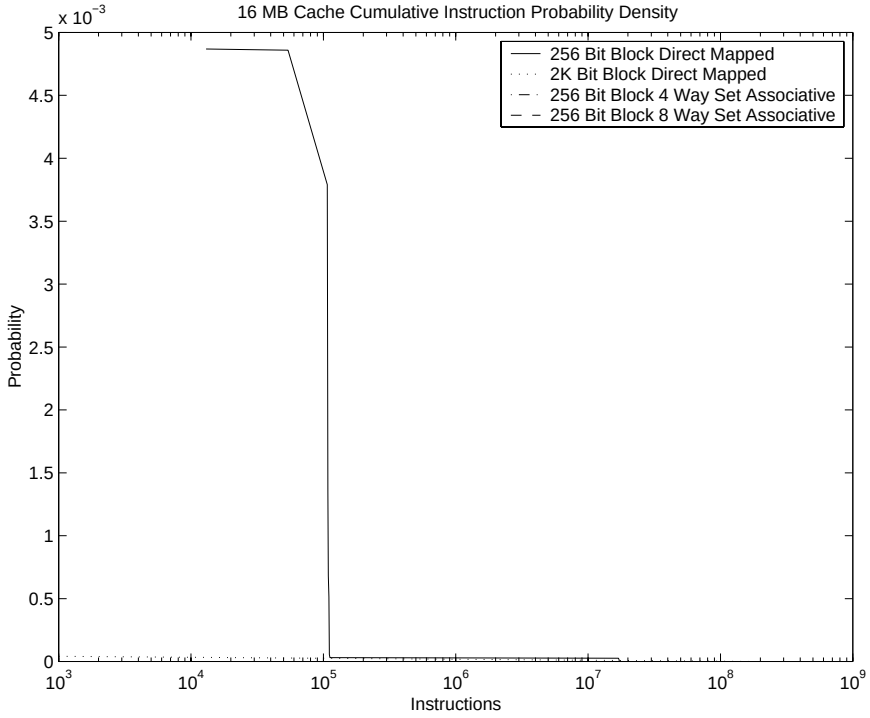


Figure B.55. DIS Image Understanding 16 MB Data Cache CIPD (Ψ)

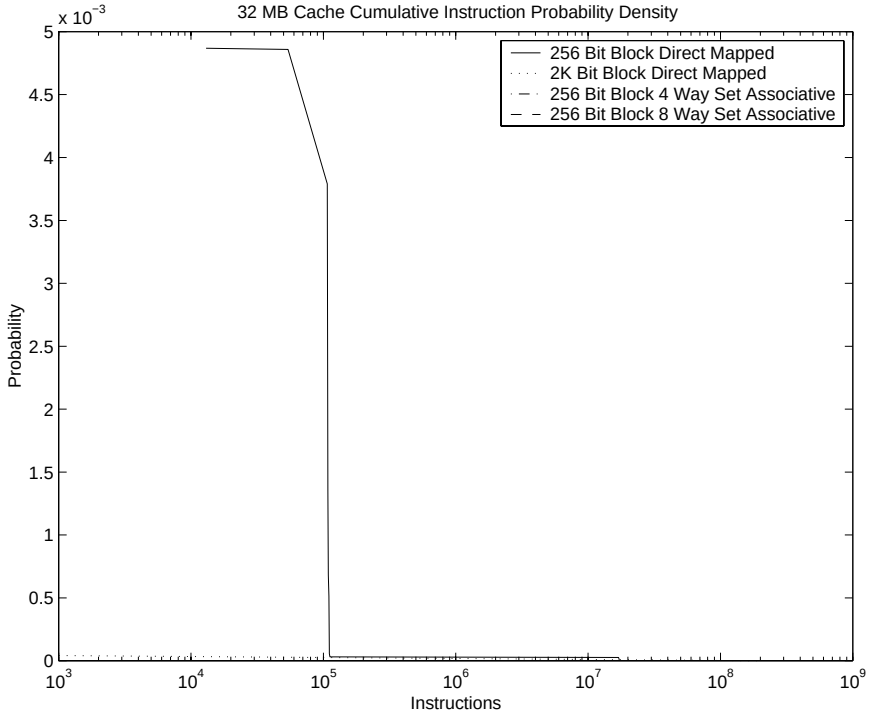


Figure B.56. DIS Image Understanding 32 MB Data Cache CIPD (Ψ)

B.5 DIS Ray Tracing

B.5.1 Page Configurations

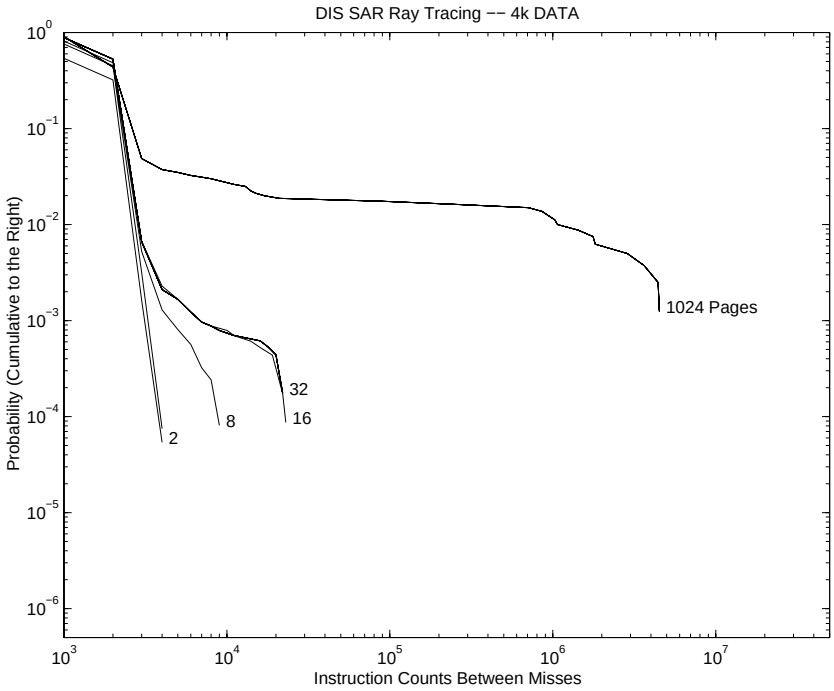


Figure B.57. DIS Ray Tracing 4 KB Page CIPD (Ψ)

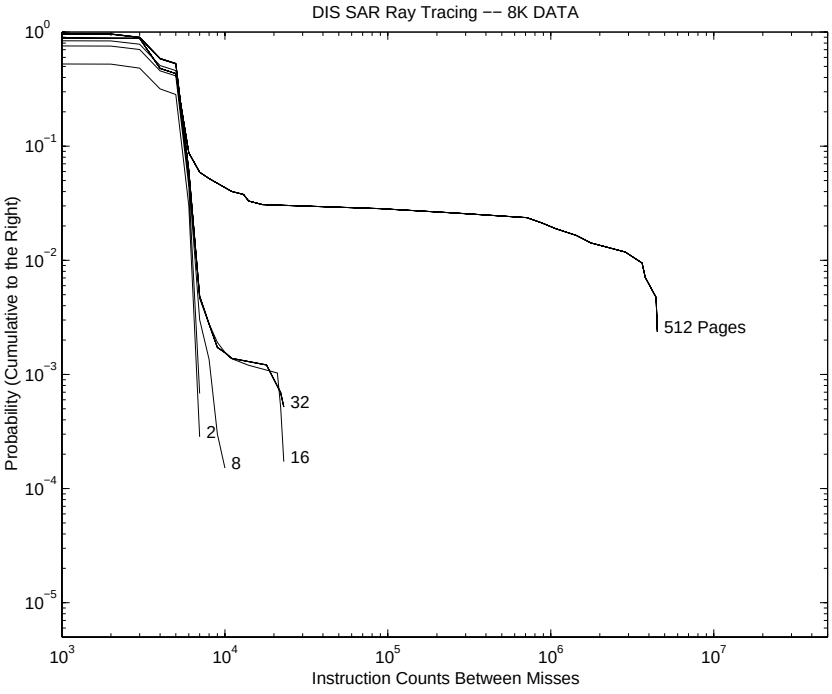


Figure B.58. DIS Ray Tracing 8 KB Page CIPD (Ψ)

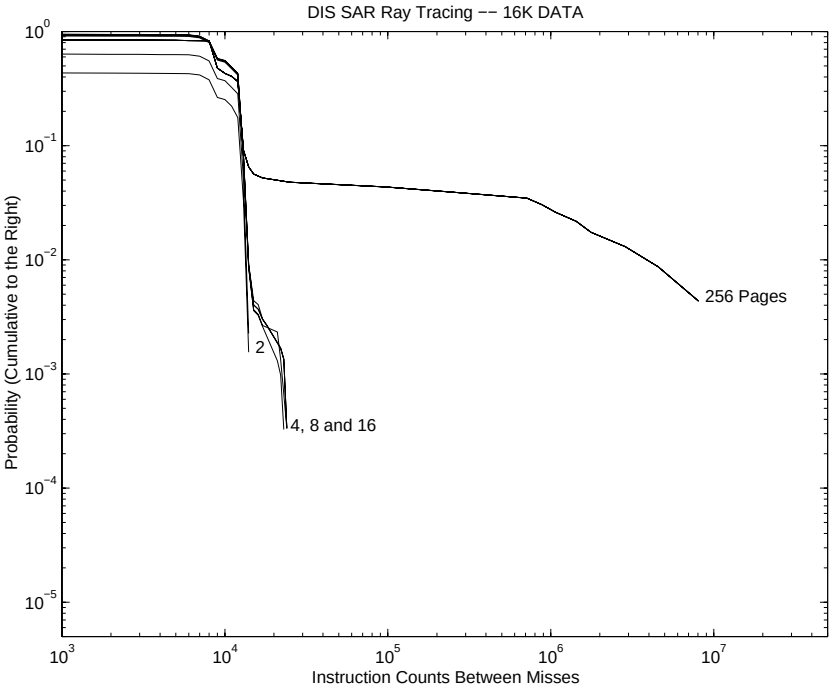


Figure B.59. DIS Ray Tracing 16 KB Page CIPD (Ψ)

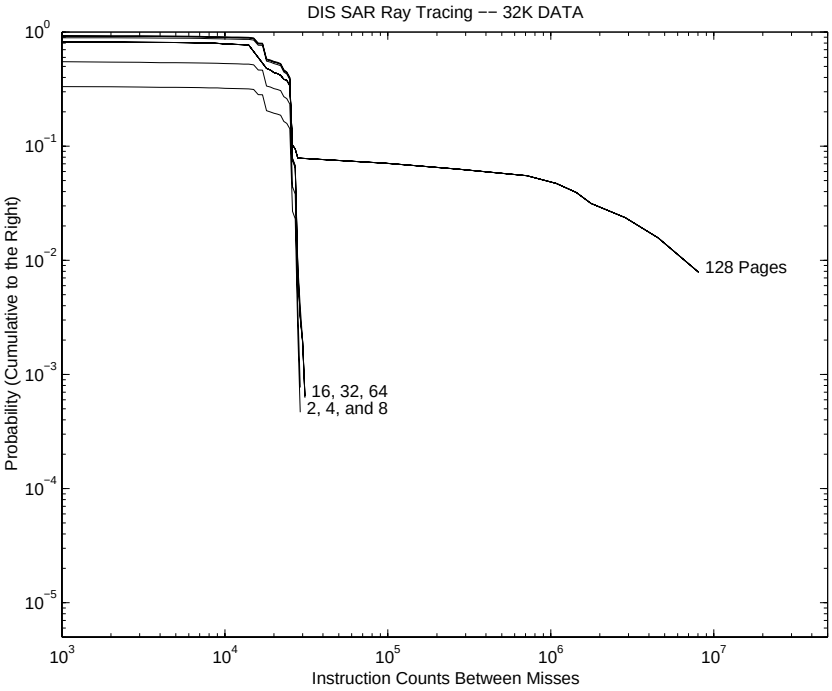


Figure B.60. DIS Ray Tracing 32 KB Page CIPD (Ψ)

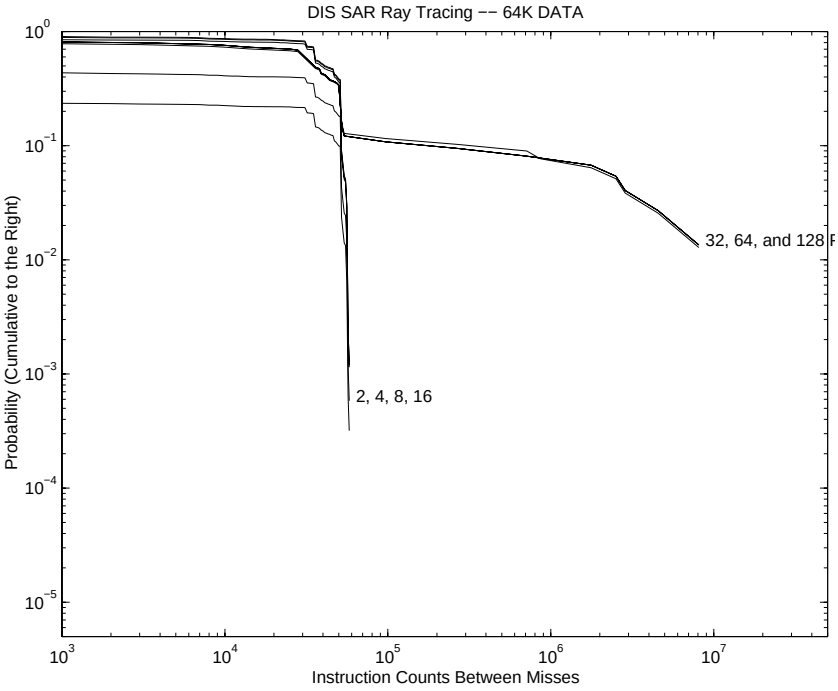


Figure B.61. DIS Ray Tracing 64 KB Page CIPD (Ψ)

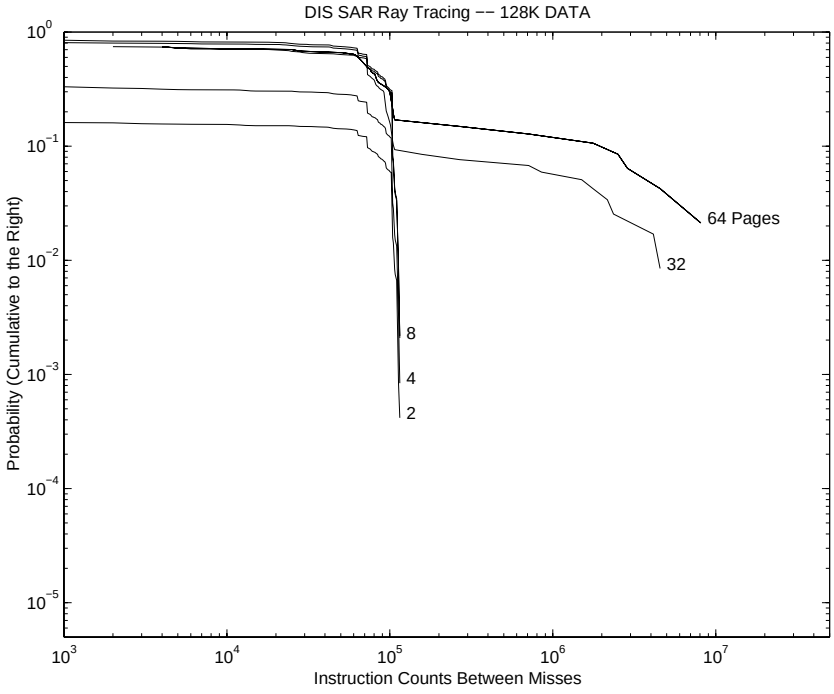


Figure B.62. DIS Ray Tracing 128 KB Page CIPD (Ψ)

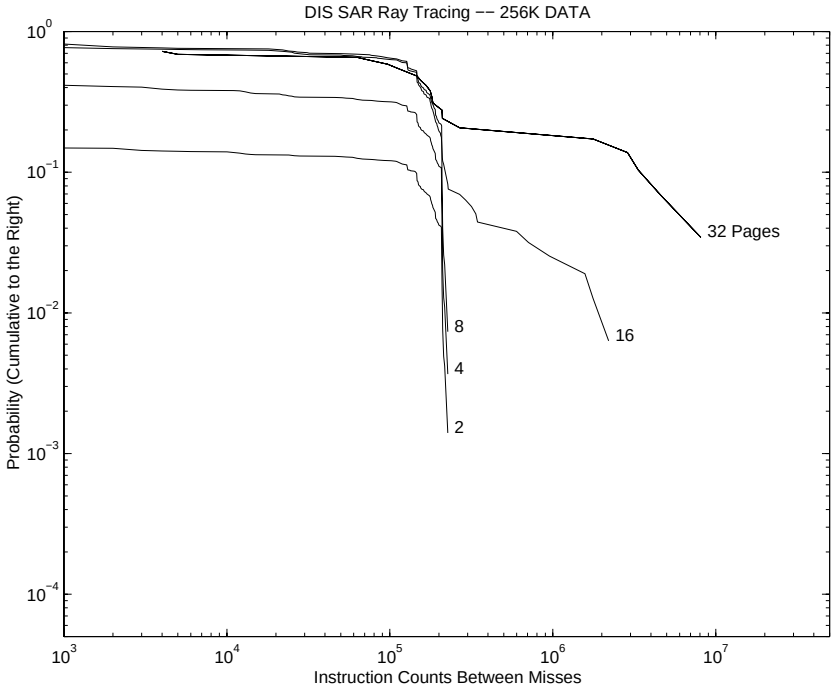


Figure B.63. DIS Ray Tracing 256 KB Page CIPD (Ψ)

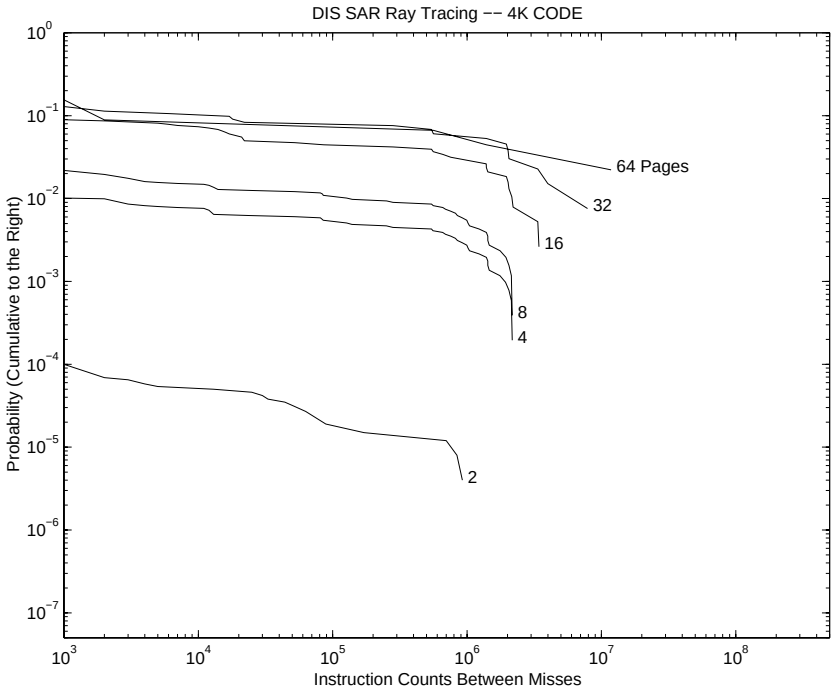


Figure B.64. DIS Ray Tracing 4 KB (Code and Stack) KB Page CIPD (Ψ)

B.5.2 Cache Configurations

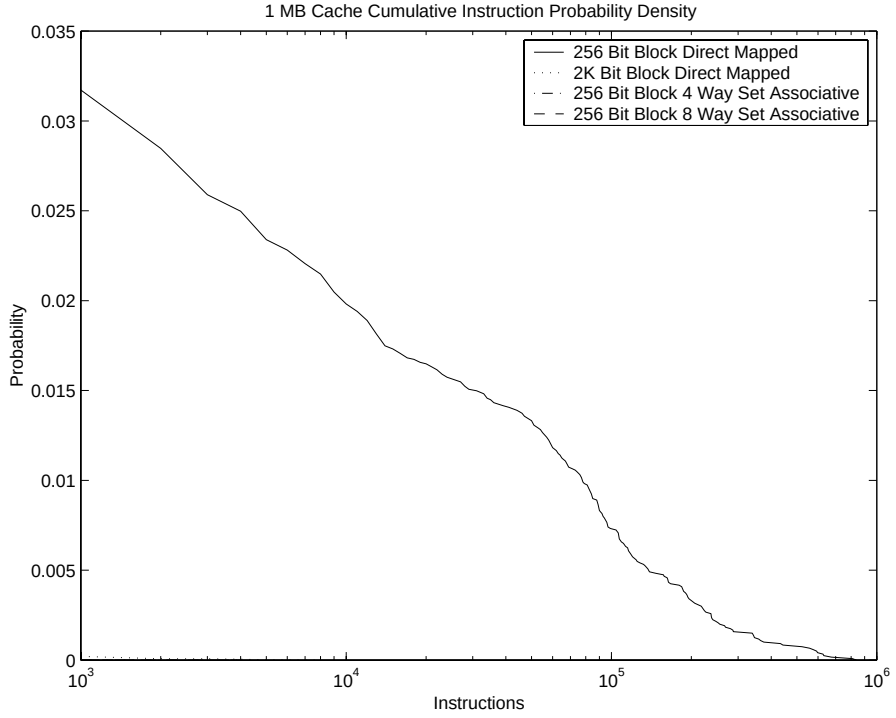


Figure B.65. DIS Ray Tracing 1 MB Data Cache CIPD (Ψ)

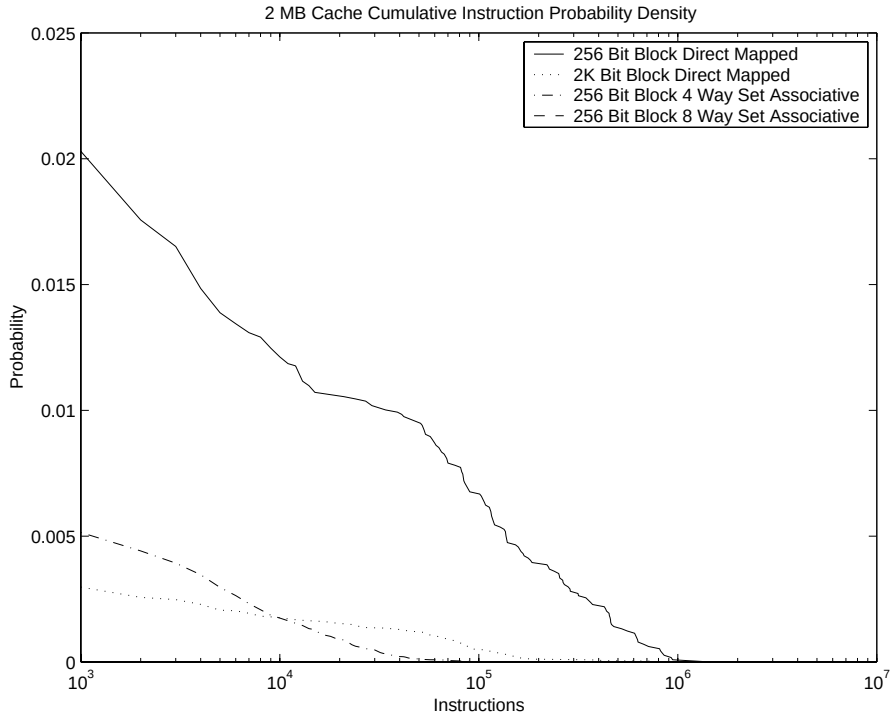


Figure B.66. DIS Ray Tracing 2 MB Data Cache CIPD (Ψ)

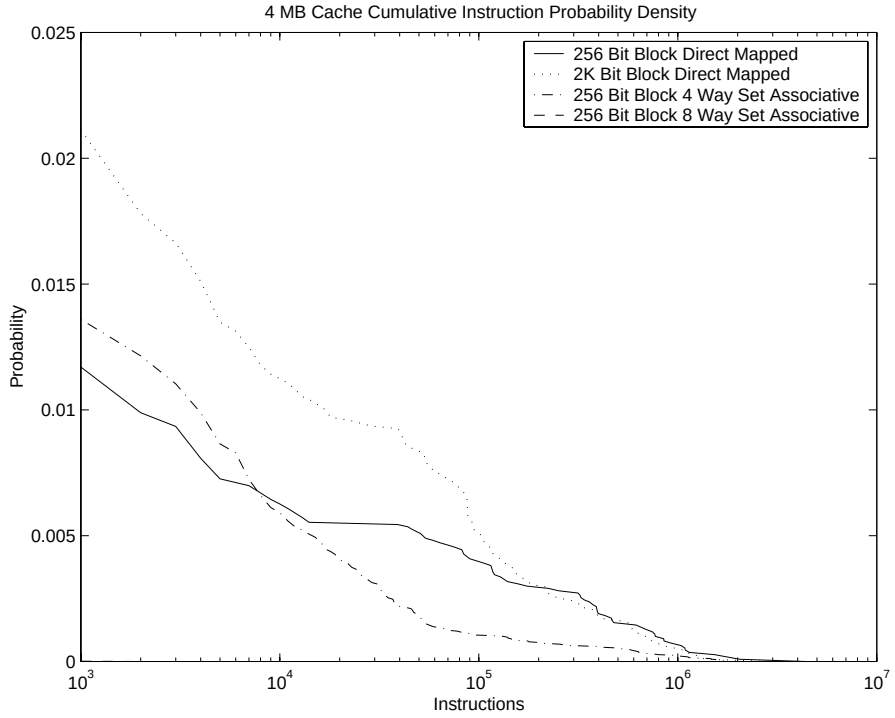


Figure B.67. DIS Ray Tracing 4 MB Data Cache CIPD (Ψ)

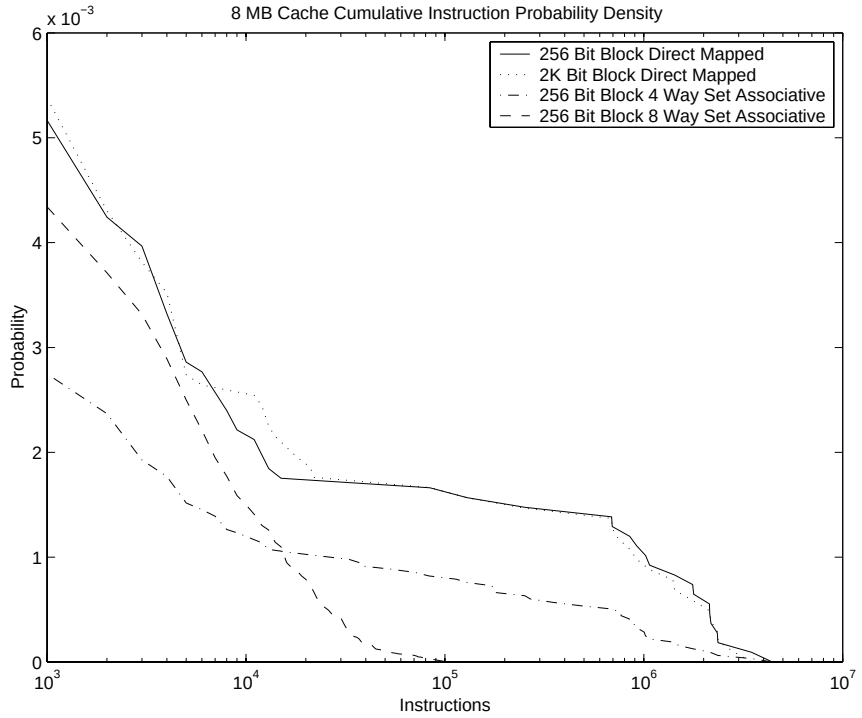


Figure B.68. DIS Ray Tracing 8 MB Data Cache CIPD (Ψ)

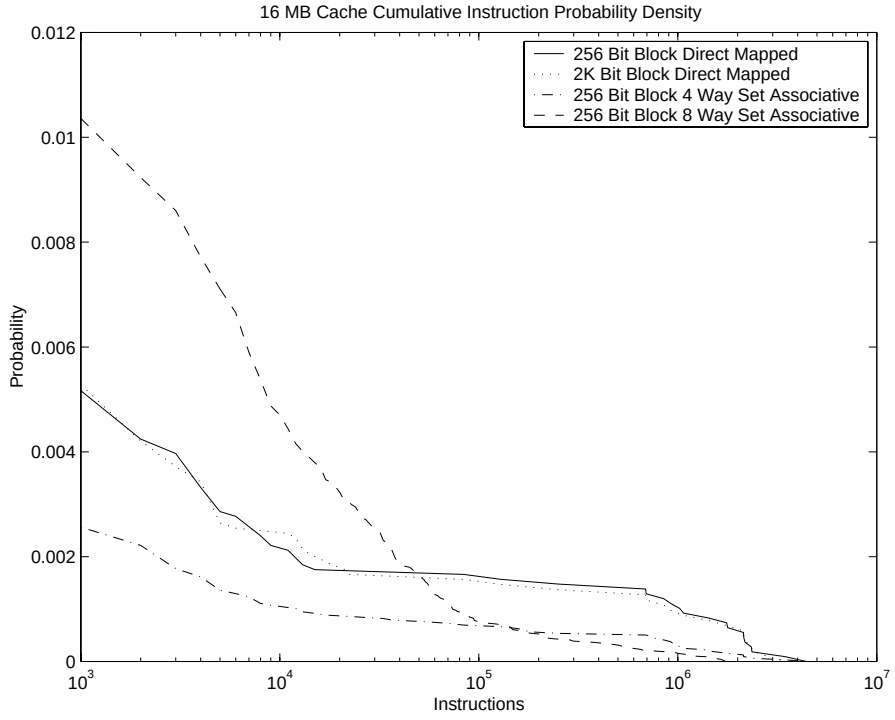


Figure B.69. DIS Ray Tracing 16 MB Data Cache CIPD (Ψ)

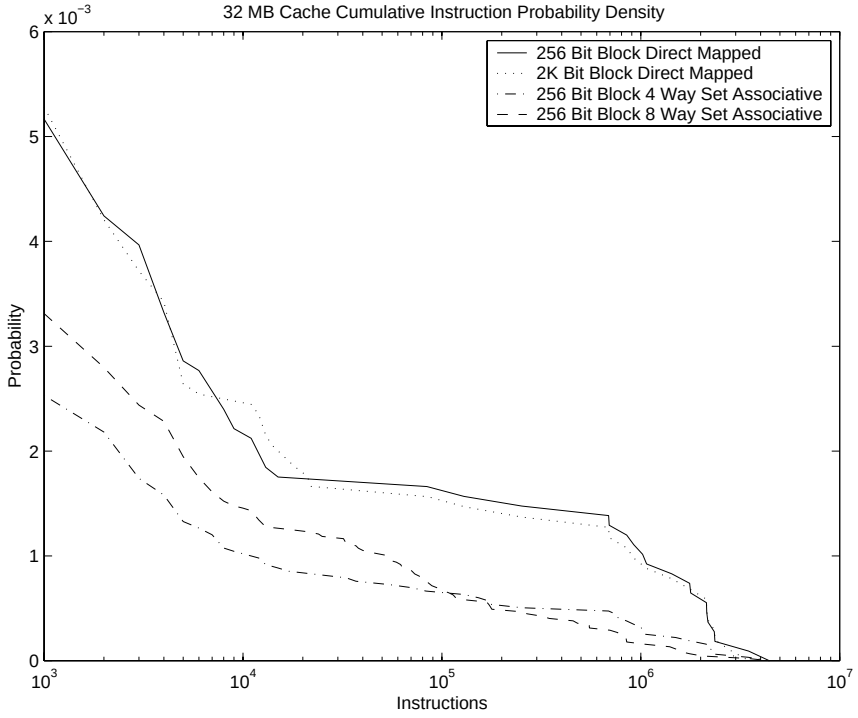


Figure B.70. DIS Ray Tracing 32 MB Data Cache CIPD (Ψ)

B.6 Molecular Dynamics Simulation

B.6.1 Page Configurations

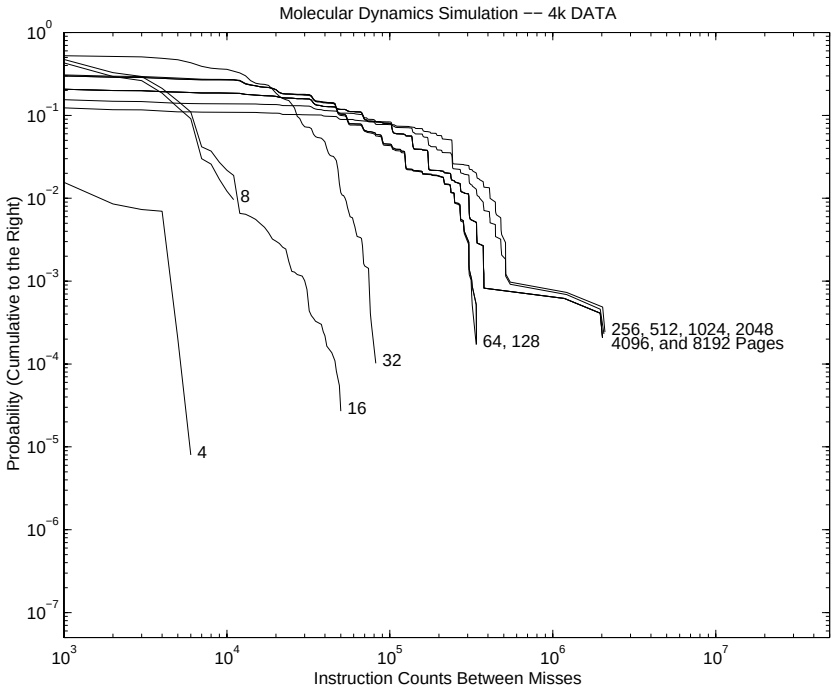


Figure B.71. Molecular Dynamics Simulation 4 KB Page CIPD (Ψ)

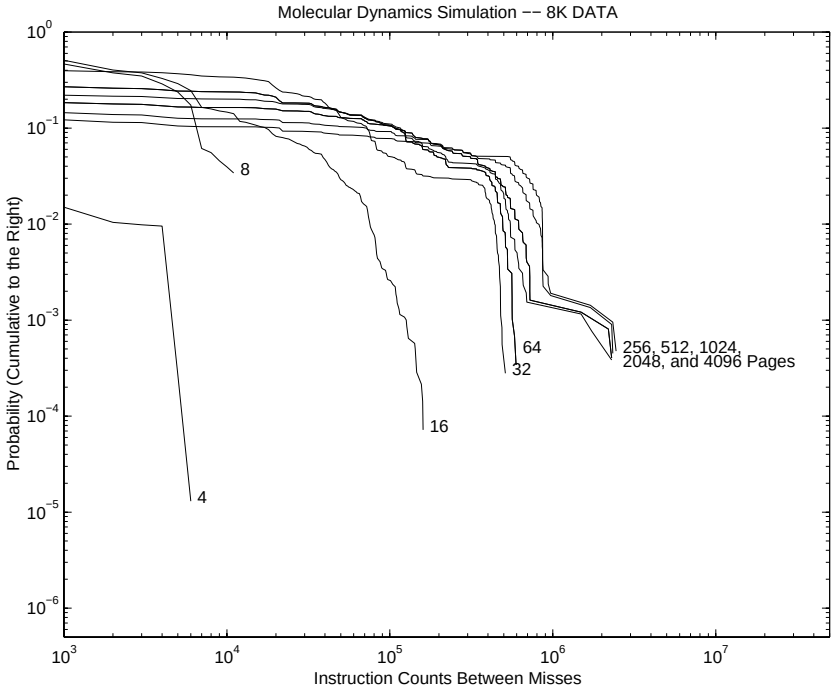


Figure B.72. Molecular Dynamics Simulation 8 KB Page CIPD (Ψ)

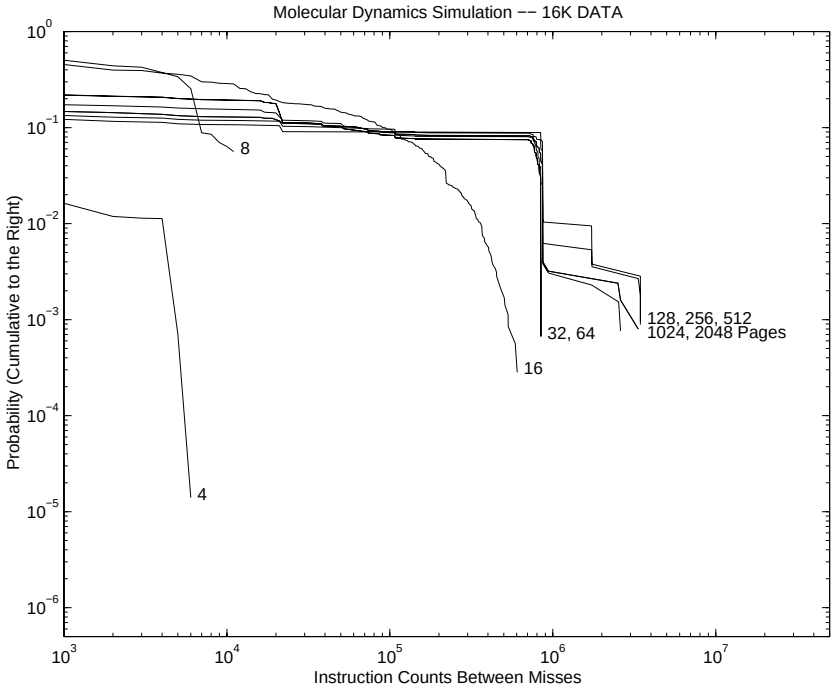


Figure B.73. Molecular Dynamics Simulation 16 KB Page CIPD (Ψ)

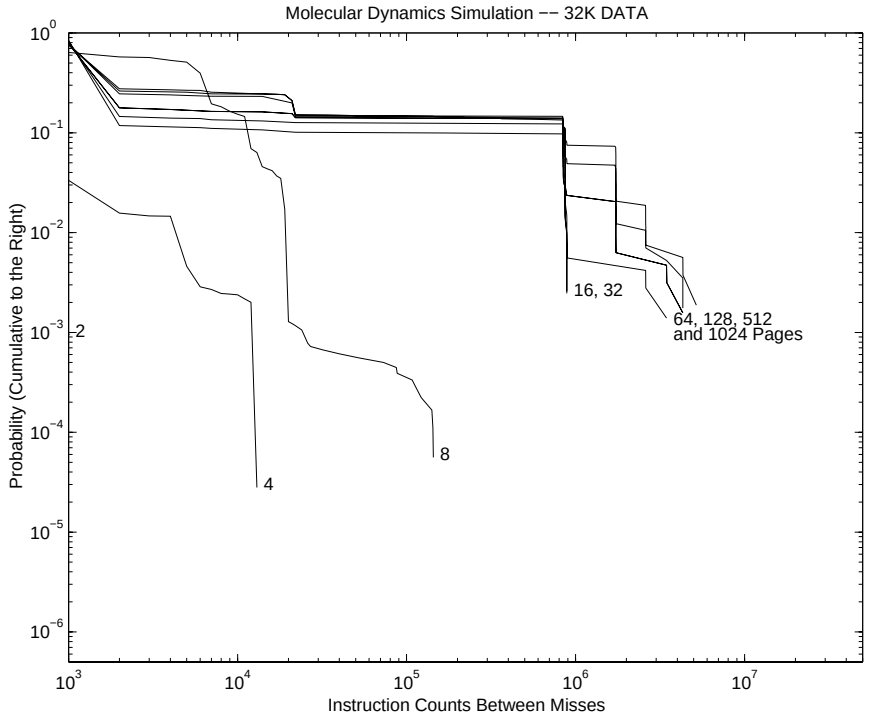


Figure B.74. Molecular Dynamics Simulation 32 KB Page CIPD (Ψ)

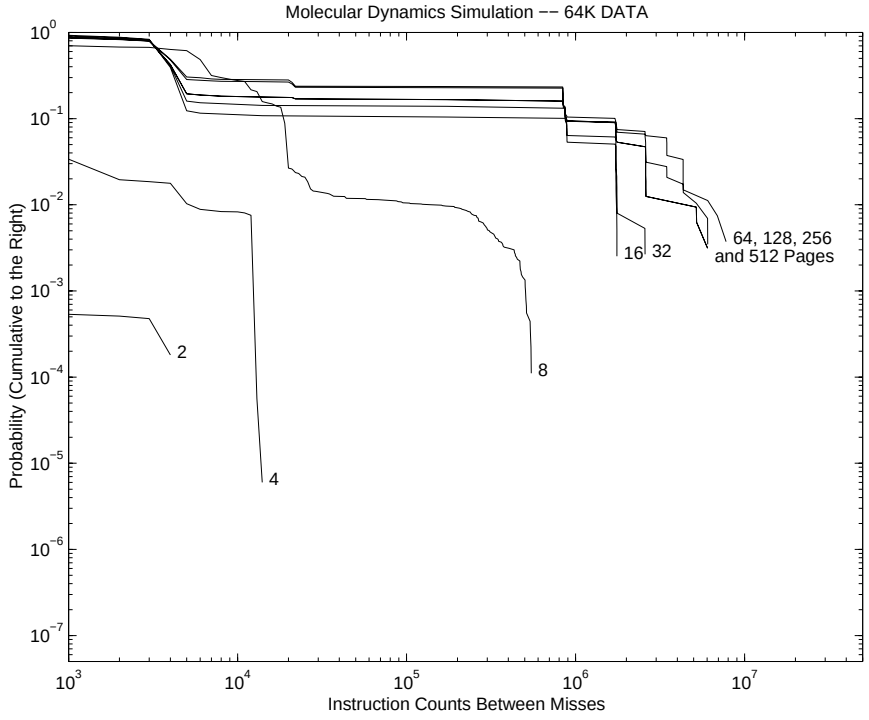


Figure B.75. Molecular Dynamics Simulation 64 KB Page CIPD (Ψ)

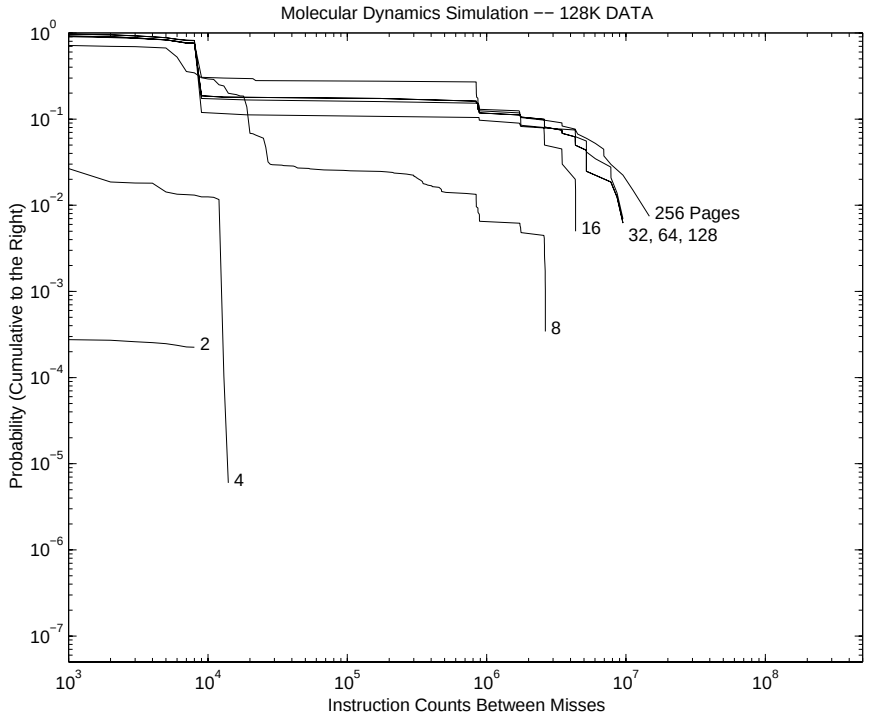


Figure B.76. Molecular Dynamics Simulation 128 KB Page CIPD (Ψ)

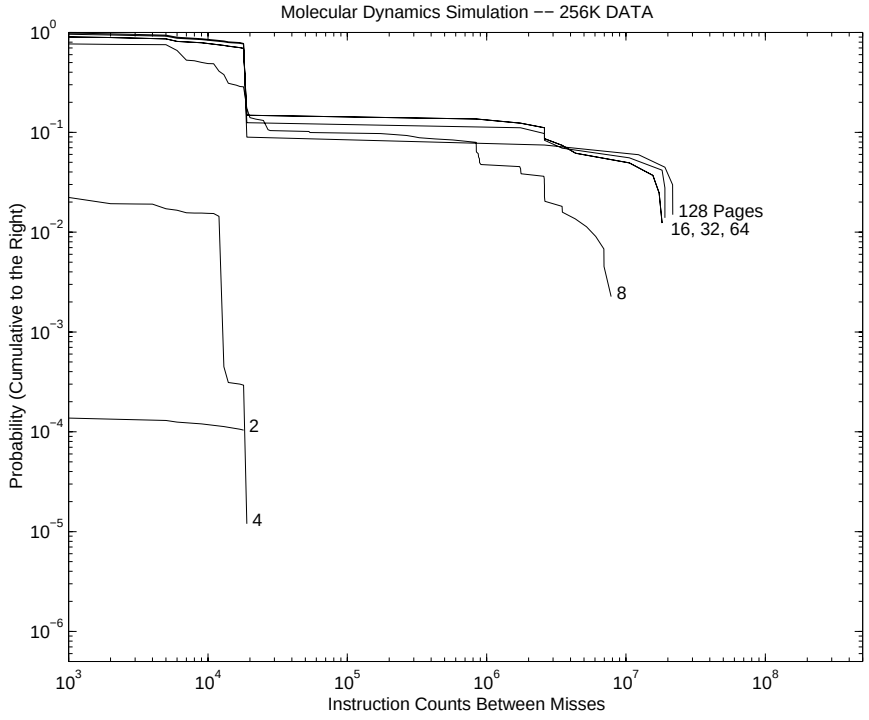


Figure B.77. Molecular Dynamics Simulation 256 KB Page CIPD (Ψ)

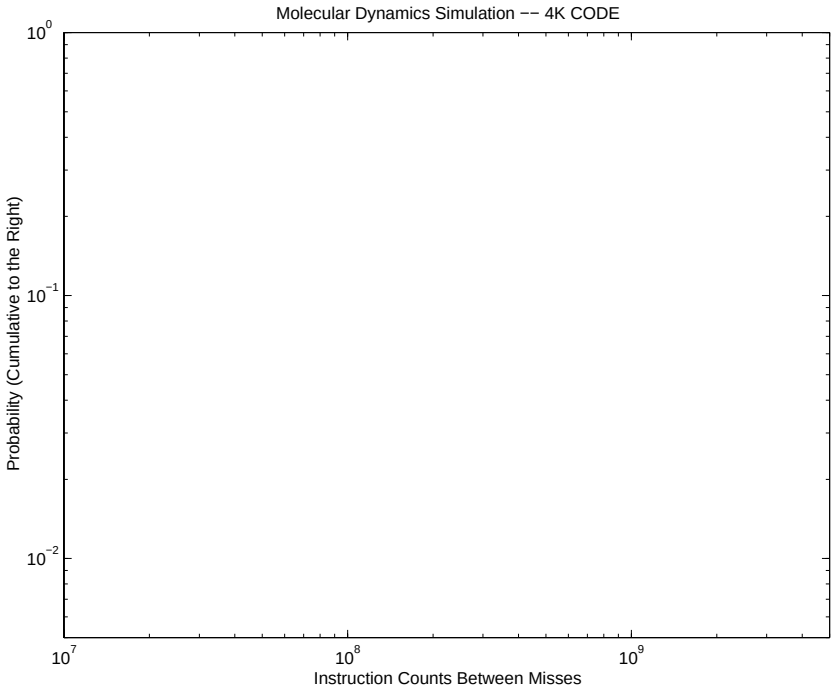


Figure B.78. Molecular Dynamics Simulation 4 KB (Code and Stack) KB Page CIPD (Ψ)

B.6.2 Cache Configurations

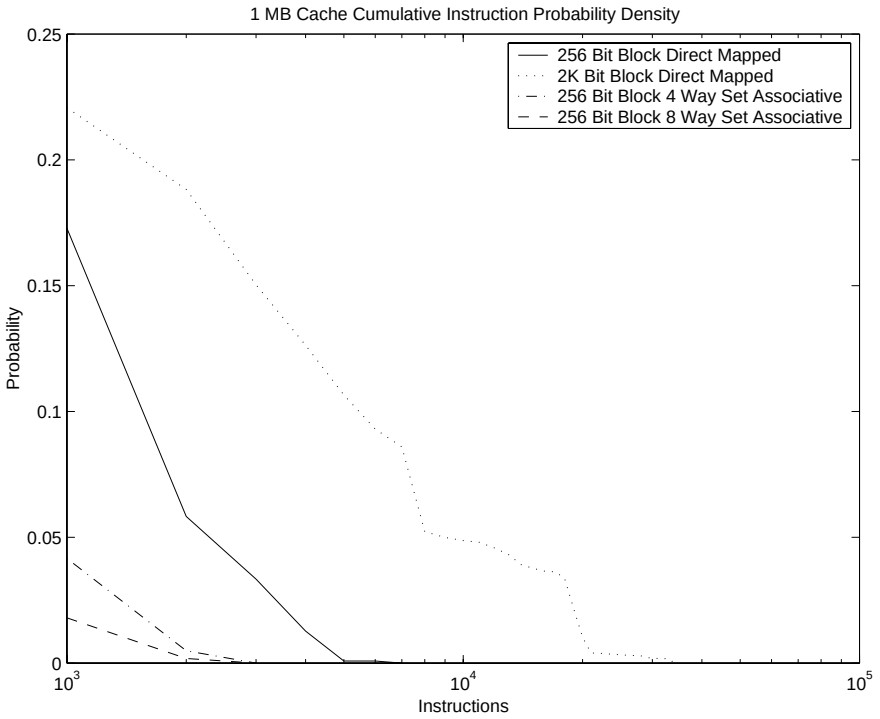


Figure B.79. Molecular Dynamics Simulation 1 MB Data Cache CIPD (Ψ)

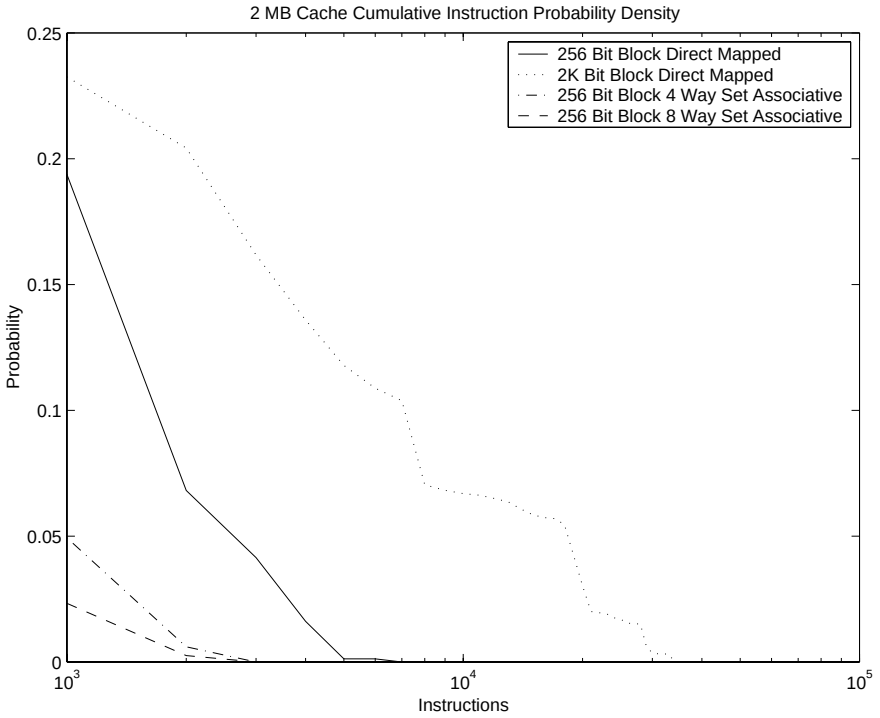


Figure B.80. Molecular Dynamics Simulation 2 MB Data Cache CIPD (Ψ)

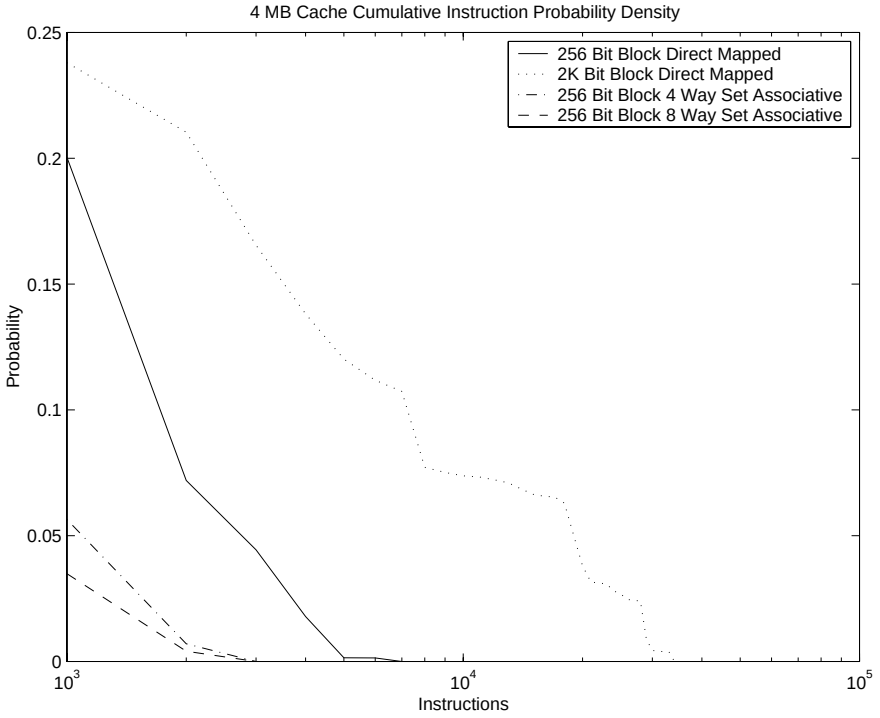


Figure B.81. Molecular Dynamics Simulation 4 MB Data Cache CIPD (Ψ)

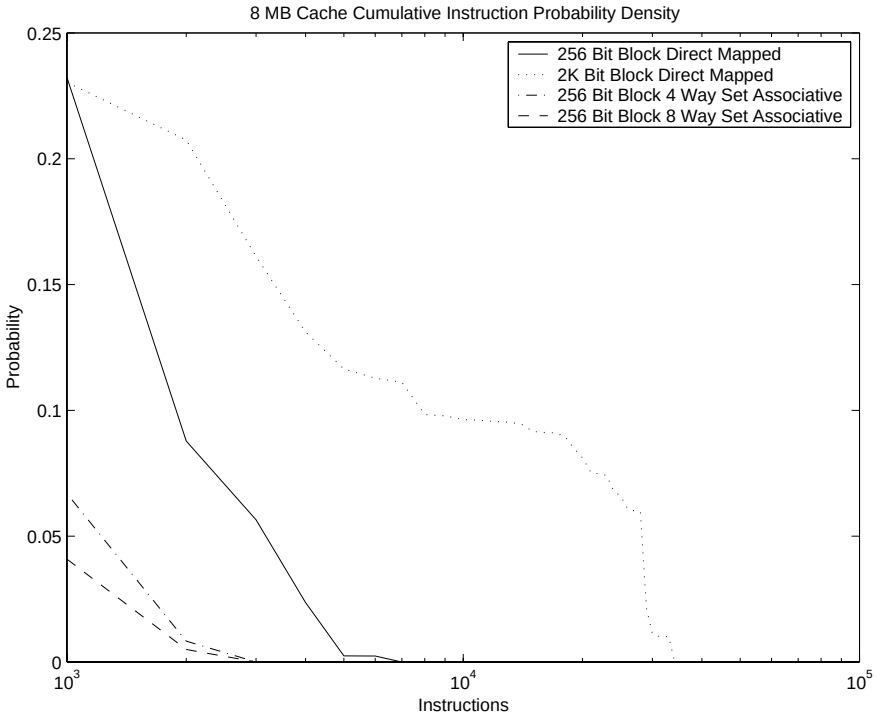


Figure B.82. Molecular Dynamics Simulation 8 MB Data Cache CIPD (Ψ)

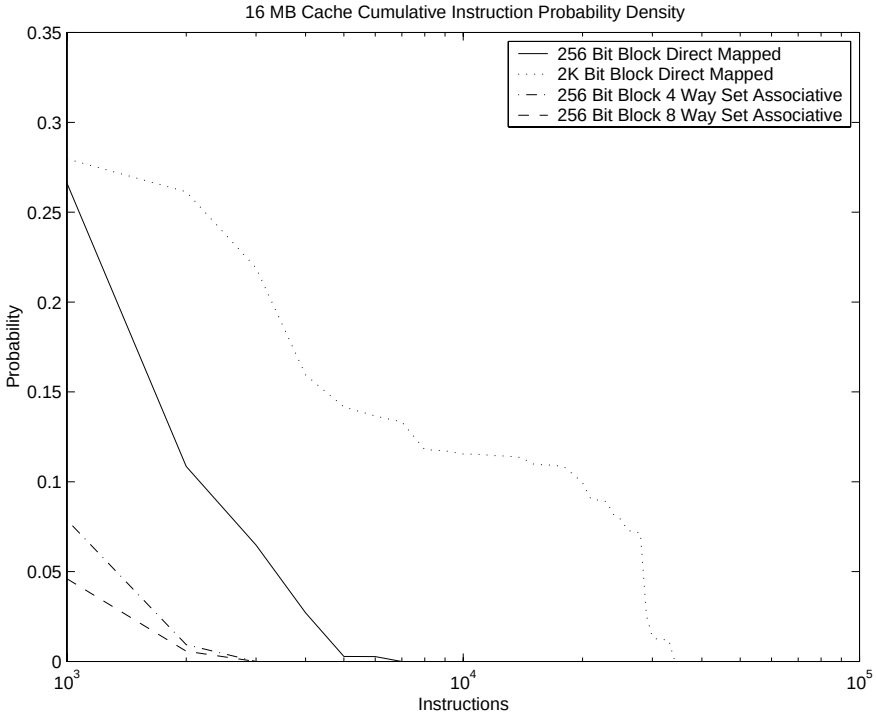


Figure B.83. Molecular Dynamics Simulation 16 MB Data Cache CIPD (Ψ)

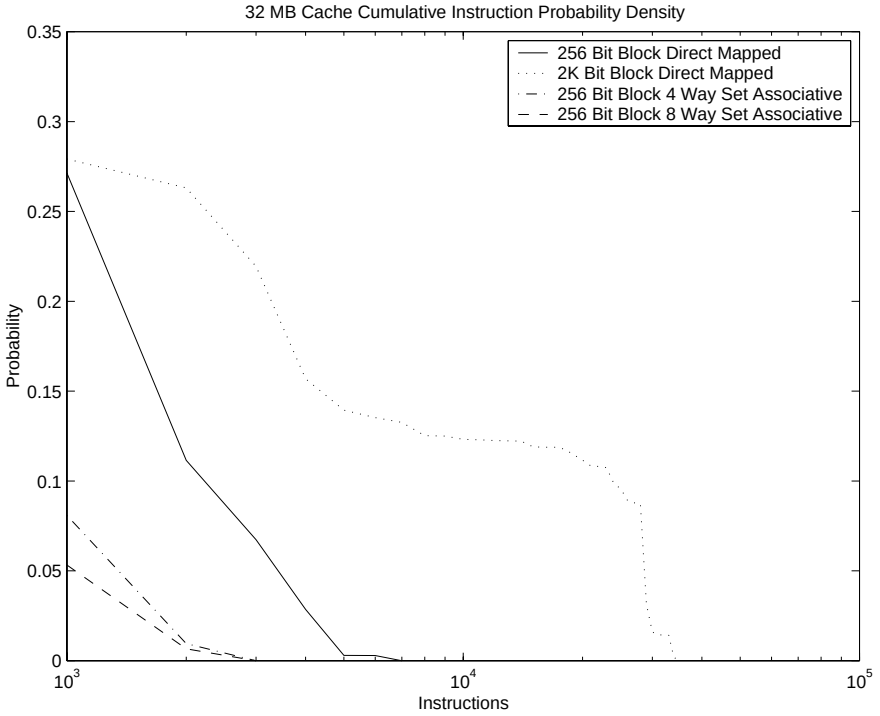


Figure B.84. Molecular Dynamics Simulation 32 MB Data Cache CIPD (Ψ)

APPENDIX C

UNABRIDGED WORKING SET PAGE MISS RATE RESULTS

C.1 DIS Data Management

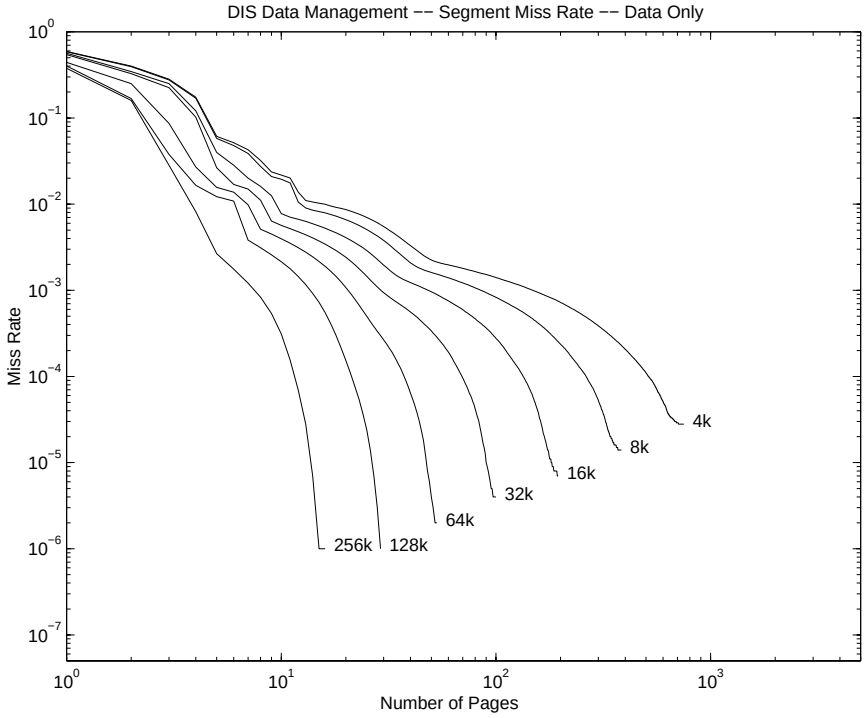


Figure C.1. DIS Data Management Data Miss Rate

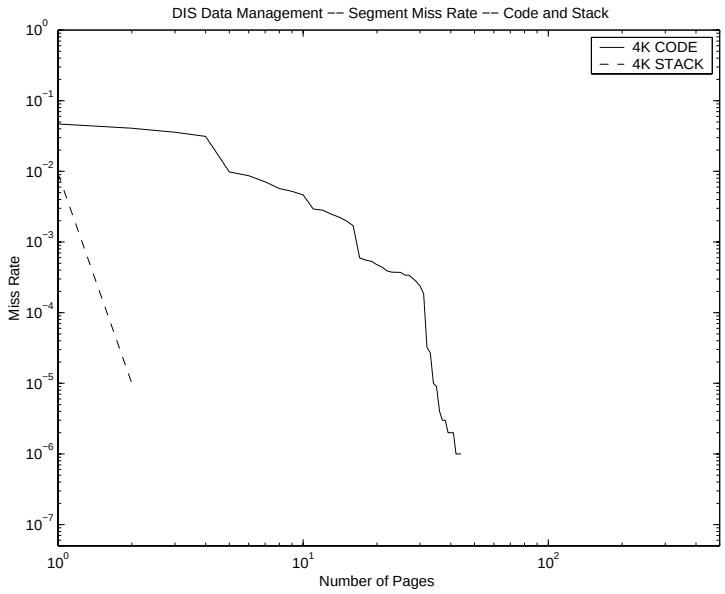


Figure C.2. DIS Data Management Code and Stack Miss Rate

C.2 DIS FFT

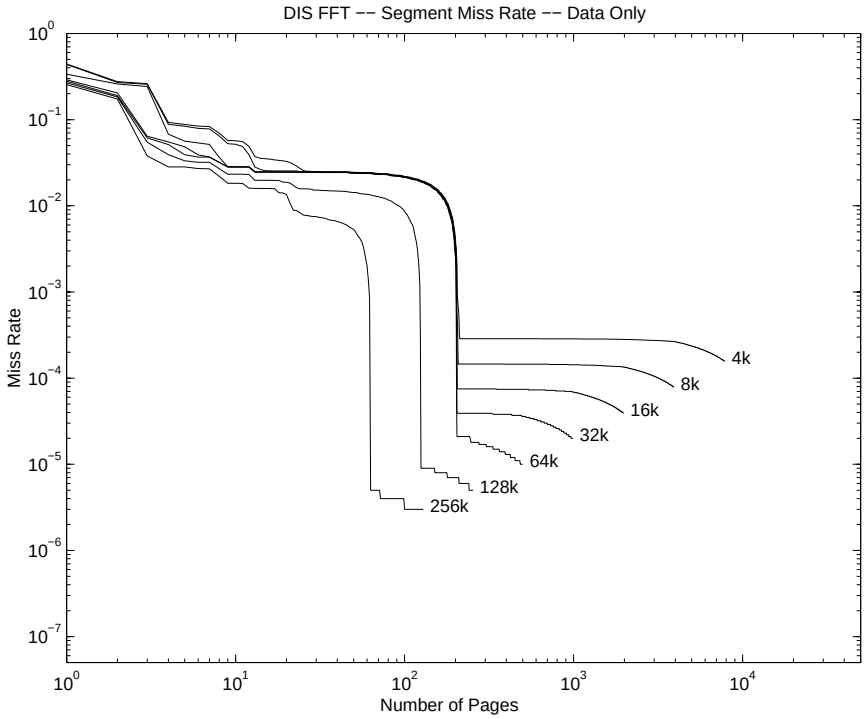


Figure C.3. DIS FFT Data Miss Rate

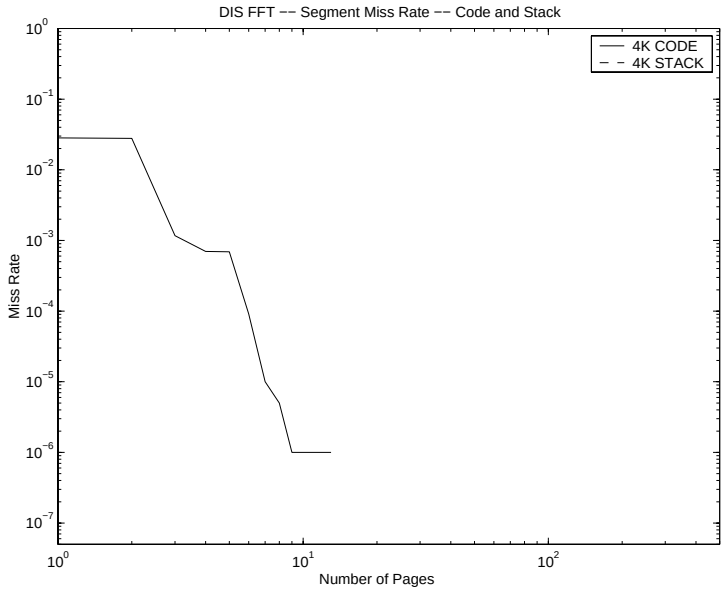


Figure C.4. DIS FFT Code and Stack Miss Rate

C.3 DIS Method of Moments

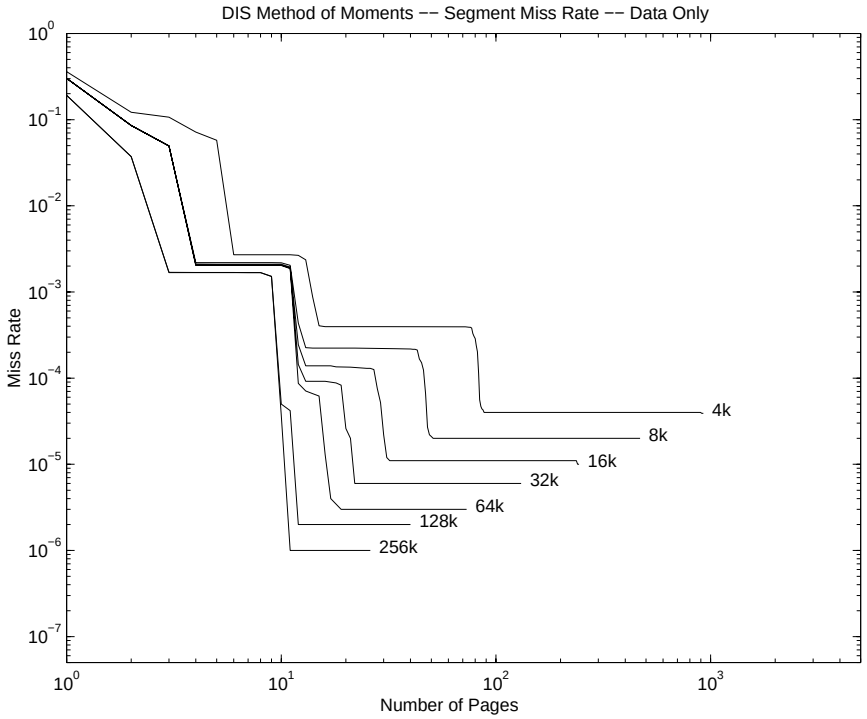


Figure C.5. DIS Method of Moments Data Miss Rate

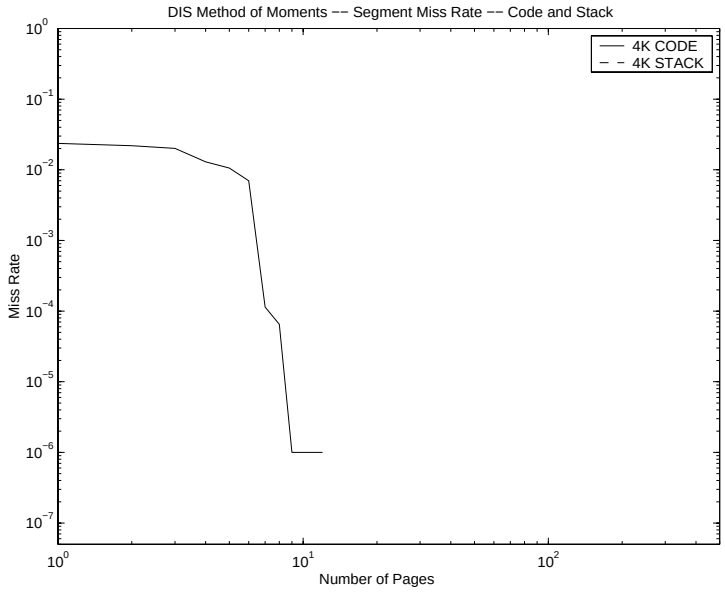


Figure C.6. DIS Method of Moments Code and Stack Miss Rate

C.4 DIS Image Understanding

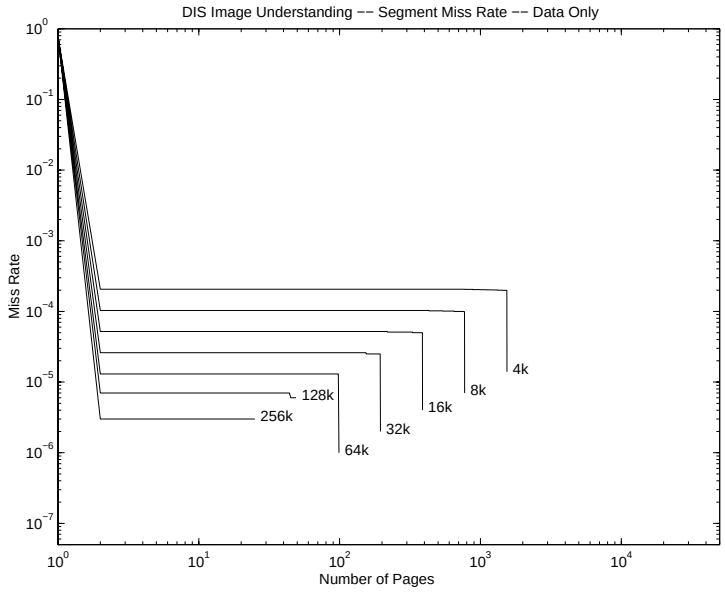


Figure C.7. DIS Image Understanding Data Miss Rate

C.5 DIS Ray Tracing

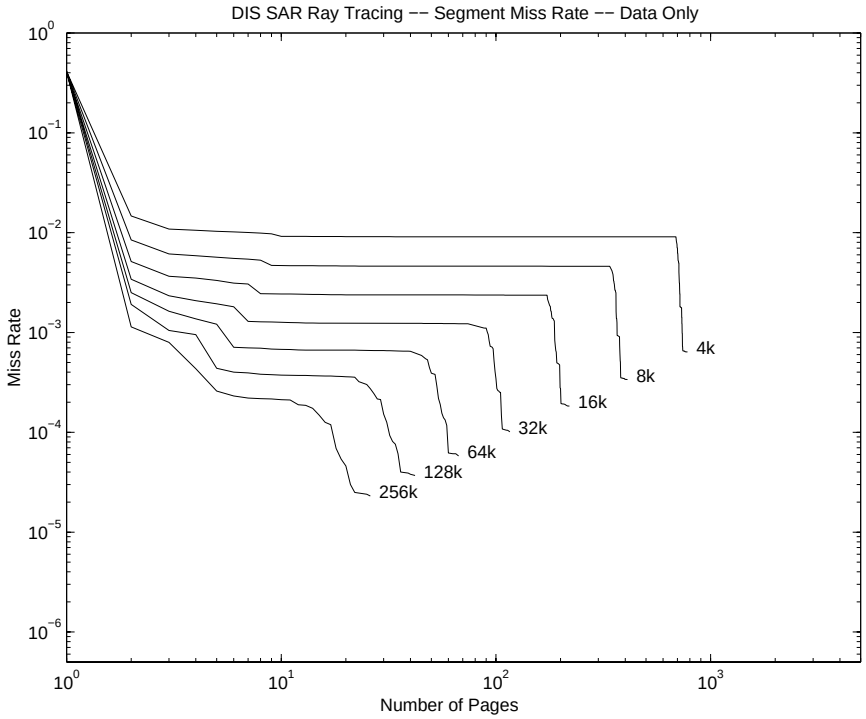


Figure C.8. DIS SAR Ray Tracing Data Miss Rate

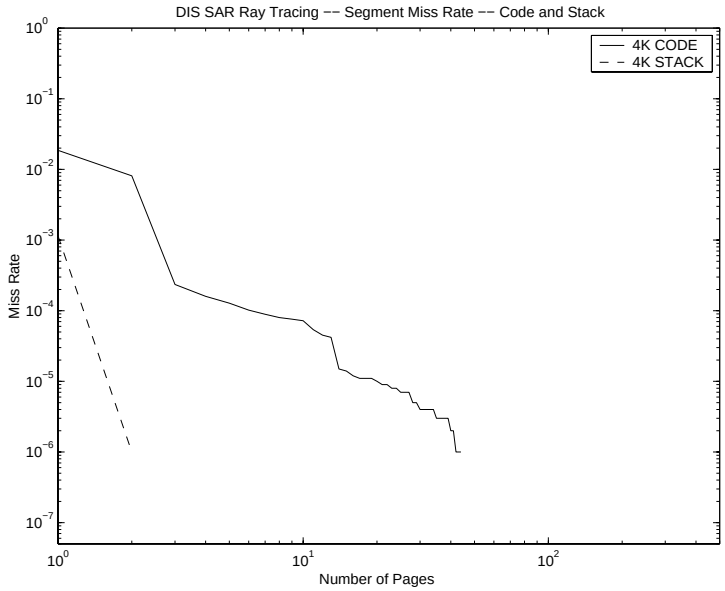


Figure C.9. DIS SAR Ray Tracing Code and Stack Miss Rate

C.6 Molecular Dynamics Simulation

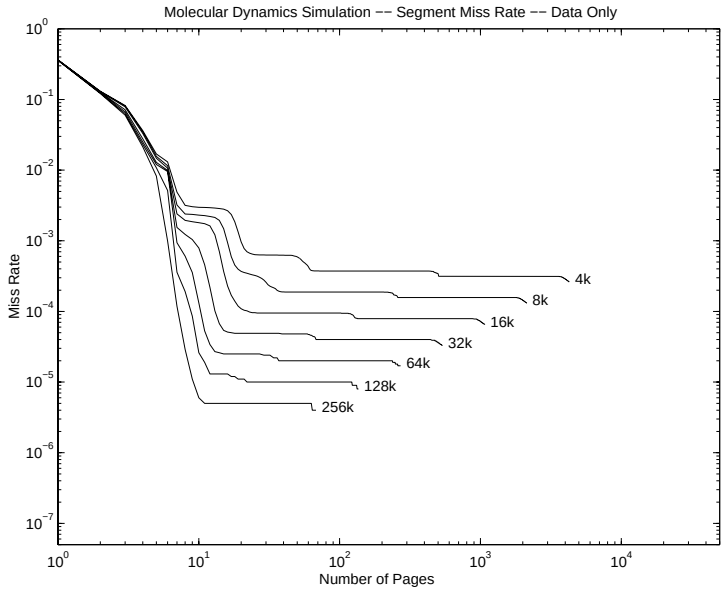


Figure C.10. Molecular Dynamics Simulation Overall Miss Rate

APPENDIX D

UNABRIDGED REMOTE NAME TRANSLATION MECHANISM RESULTS

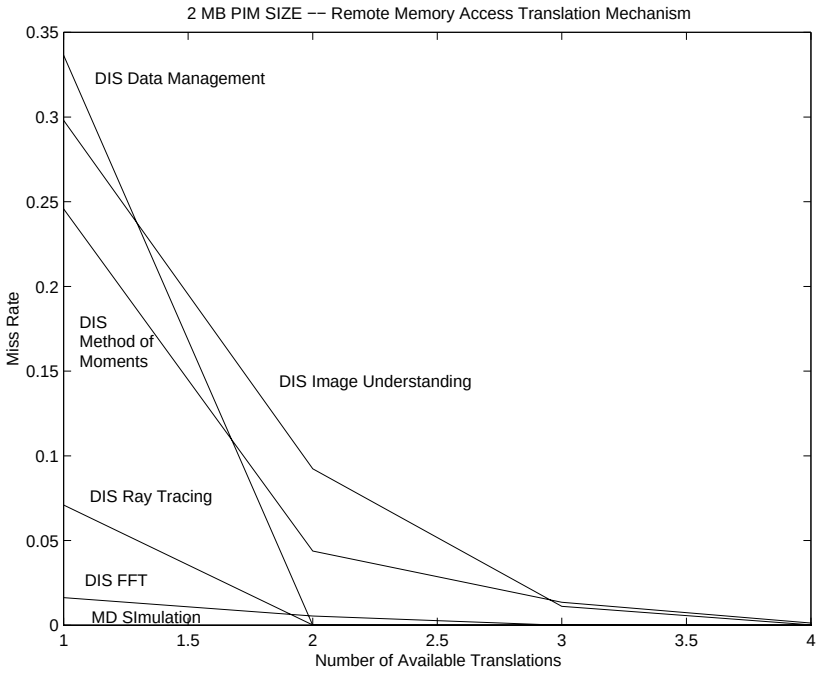


Figure D.1. Remote Name Translation Mechanism Miss Rate (2 MB PIM)

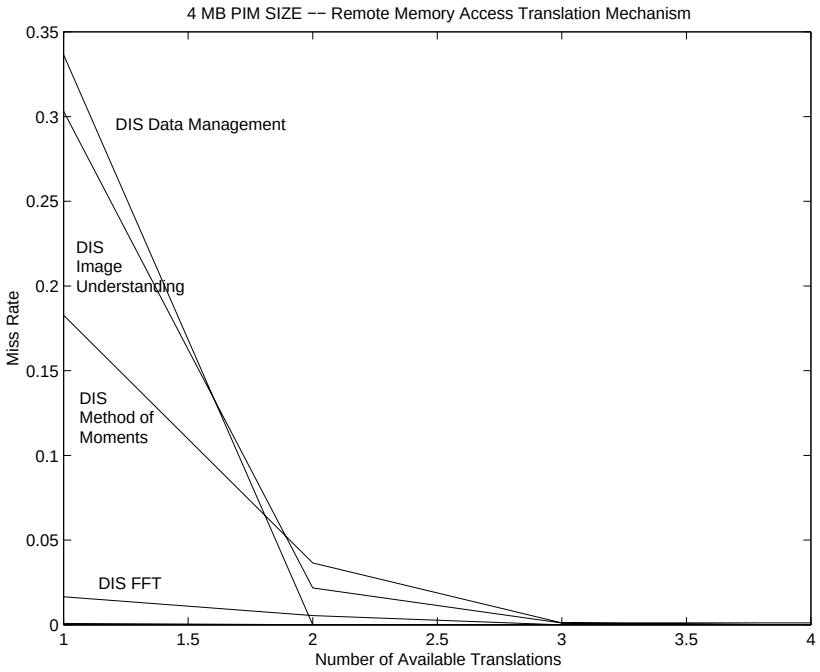


Figure D.2. Remote Name Translation Mechanism Miss Rate (4 MB PIM)

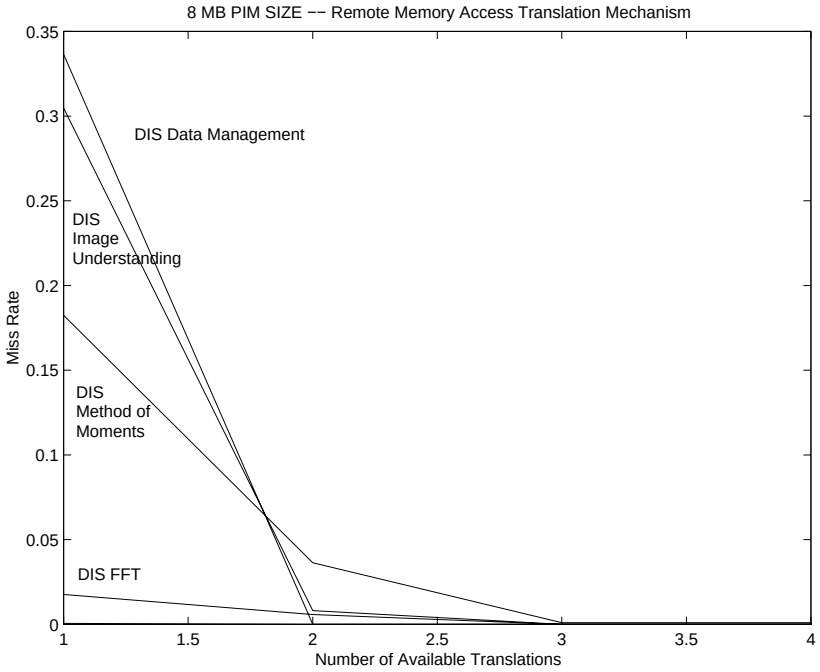


Figure D.3. Remote Name Translation Mechanism Miss Rate (8 MB PIM)

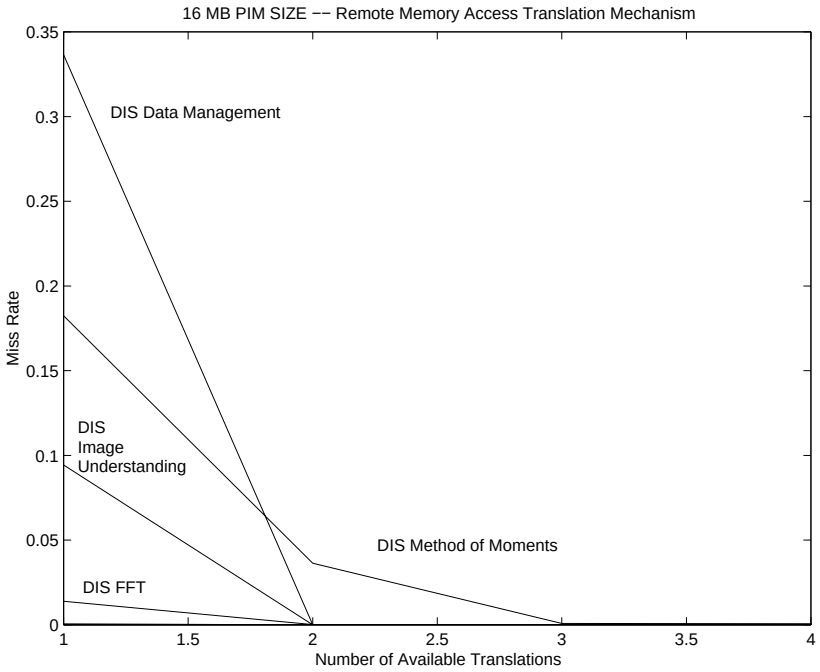


Figure D.4. Remote Name Translation Mechanism Miss Rate (16 MB PIM)

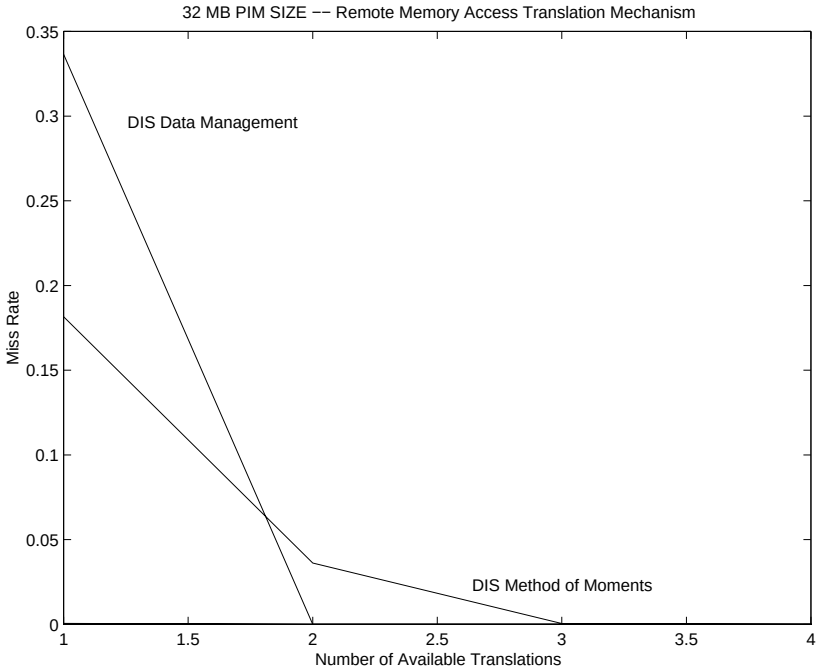


Figure D.5. Remote Name Translation Mechanism Miss Rate (32 MB PIM)

APPENDIX E

UNABRIDGED CARPET BAG CACHE MISS RATE RESULTS FOR THE IMAGE UNDERSTANDING BENCHMARK

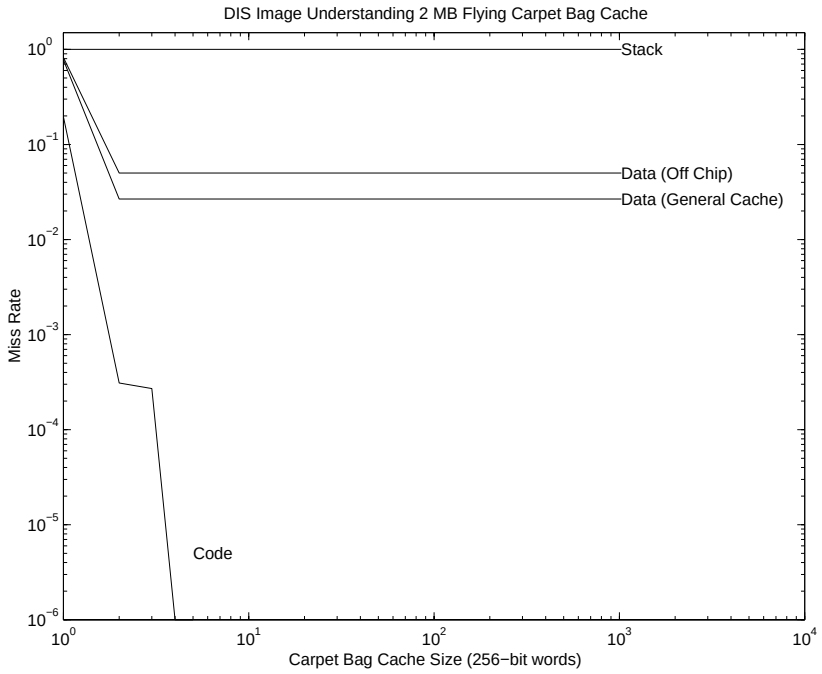


Figure E.1. DIS Image Understanding Carpet Bag Cache Miss Rate (2 MB)

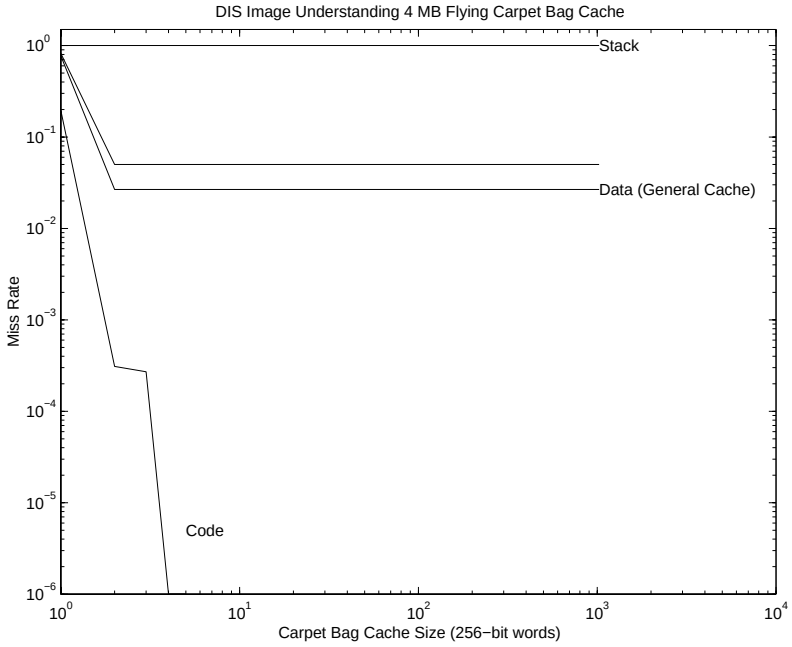


Figure E.2. DIS Image Understanding Carpet Bag Cache Miss Rate (4 MB)

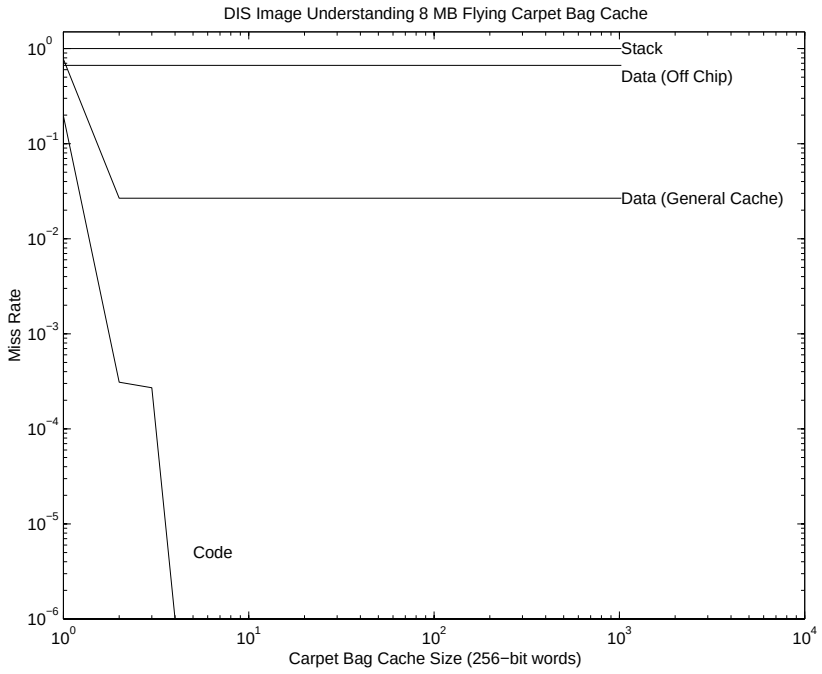


Figure E.3. DIS Image Understanding Carpet Bag Cache Miss Rate (8 MB)

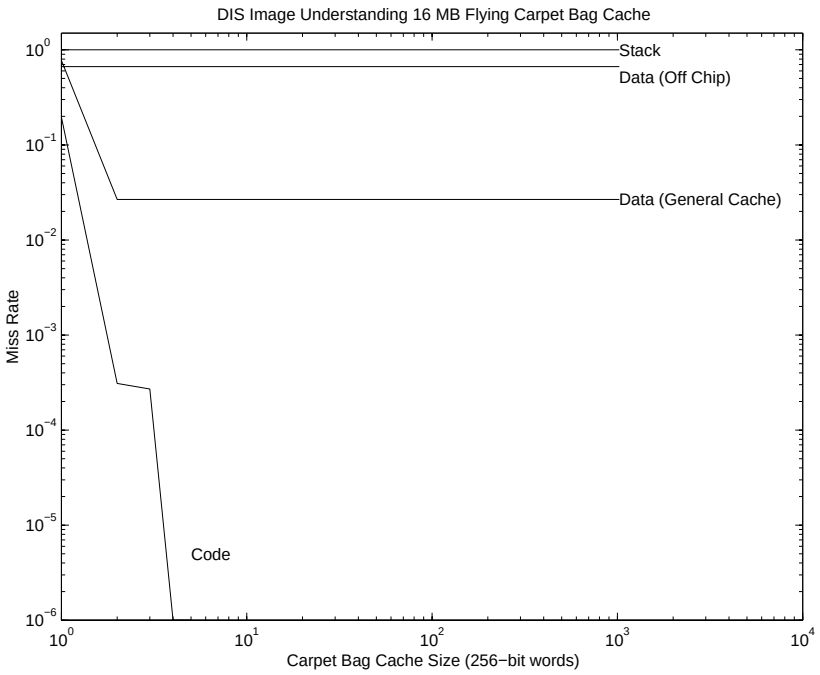


Figure E.4. DIS Image Understanding Carpet Bag Cache Miss Rate (16 MB)

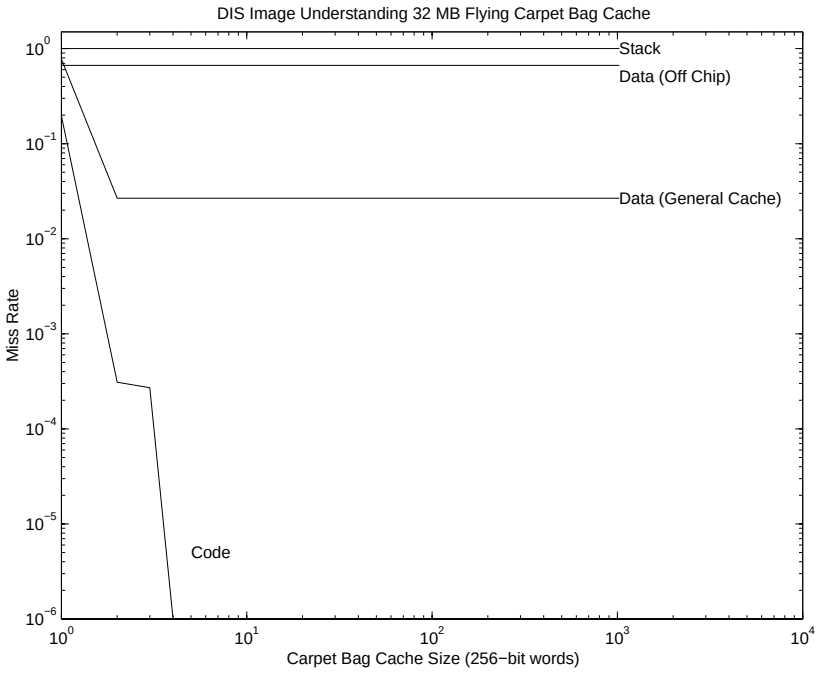


Figure E.5. DIS Image Understanding Carpet Bag Cache Miss Rate (32 MB)

BIBLIOGRAPHY

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, , and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. *Proceedings of the 22nd International Symposium on Computer Architecture.*

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera System. *Tera Computer Company.*

[3] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer, Vol. 29, No. 2*, February 1996.

[4] Atlantic Aerospace Electronics Corporation. *Data-Intensive Systems Benchmark Suite Analysis and Specification*, 1.0 edition, June 1999.

[5] Doug Burger. System-Level Implications of Processor-Memory Integration. *Proceedings of the 24th International Symposium on Computer Architecture*, June, 1997.

[6] Doug Burger, James R. Goodman, and Alain Kagi. Limited Bandwidth to Affect Processor Design. *IEEE Micro*, November/December 1997.

[7] Doug Burger and Alain Kagi. Memory bandwidth Limitations of Future Microprocessors. *Proceedings of the 23th International Symposium on Computer Architecture*, May, 1996.

[8] Michael J. Carey, David J Dewitt, and Jeffery F. Naughton. The oo7 benchmark. In *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*, 1993.

[9] David Culler, Kim Keeton, Cedric Krumbein, Lok Tin Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, and Chad Yoshikawa. Generic Active Message Interface Specification. February 1995.

[10] W.J. Dally, J. Fiske, J Keene, R. Lethin, M. Noakes, P. Nuth, R. Davidson, and G. Fyler. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, April 1992.

[11] Stefanos Damianakis, Angelos Bilas, Cezary Dubnicki, and Edward W. Felten. Client Server Computing on the SHRIMP Multicomputer. *IEEE Micro*, February 1997.

[12] B Dembart and E.L. Yip. A 3-d Fast Multipole Method for Electromagnetics with Multiple Levels. *ISSTECH-97-004, The Boeing Company*, December 1994.

[13] Duhamel and Vetterli. Fast Fourier Transforms: a Tutorial Review and State of the Art. *Signal Processing*, 19, April 1990.

[14] M.A. Epton and B Dembart. Low Frequency Multipole Translation for the Helmholtz Equation. *SSGTECH-98-013, The Boeing Company*, August 1994.

[15] M.A. Epton and B Dembart. Multipole Translation Theory for the 3-d Laplace and Helmholtz Equations. *SIAM Journal of Scientific Computing*, 16(4), July 1995.

[16] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *Technical Report No. CS-93-214 (revised), University of Tennessee*, April 1994.

[17] Guttman. R-Trees: a Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOID*, June 1984.

[18] Linley Gwennep. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, October 28, 1998.

[19] J. M. Haile. *Molecular Dynamics Simulation : Elementary Methods*. John Wiley & Sons, 1997.

[20] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill Series in Computer Science, 1993.

[21] IEEE Standard 1596-1992. The SCI Standard.

[22] Kogge, Peter M. and et al. *Final Report: PIM Architecture Design and Supporting Trade Studies for the HTMT Project*, September 1999.

[23] Banks Kornacker. High-Concurrency Locking in R-Tree. In *Proceedings of 21st International Conference on Very Large Data Bases*, September 1995.

[24] J. T. Kuehn and B. J. Smith. The Horizon Supercomputer System: Architecture and Software. *Proceedings of Supercomputing 1988*, November 1988.

[25] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, , and John Hennessy. The Stanford FLASH Multiprocessor. *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[26] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Implementation and Performance. *19th International Symposium on Computer Architecture*, 1993.

[27] Notre Dame PIM Development Group. *ASAP Principles of Operation*, February 2000.

[28] SPEC Open Systems Steering Committee. Spec run and reporting rules for cpu95 suites. September 11, 1994.

[29] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: A Computation Model for Intelligent Memory. *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[30] David Patterson, Thomans Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April, 1997.

[31] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, 2ed.* Morgan Kaufmann Publishers, 1997.

[32] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. An Argument For Simple COMA. *First IEEE Symposium on High Performance Computer Architecture*, January 1995.

[33] Artour Stoutchinin, José Nelson Amaral, Guang R. Gao, Jim Dehnert, and Suneel Jain. Automatic Prefetching of Induction Pointers for Software Pipelining. *CAPSL Technical Memo, University of Deleware*, November 1999.

[34] Sun Microsystems. *Introduction to Shade*, June 1997.

[35] M. R. Thistle and B. J. Smith. A Processor Architecture for Horizon. *Proceedings of Supercomputing 1988*, November 1988.

[36] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*. ACM Press, 1992.

[37] David H.D. Warren and Seif Haridi. The Data Diffusion Machine – A Scalable Shared Virtual Memory Multiprocessor. *1988 International Conference on Fifth Generation Computer Systems*.